Monadification, Memoization and Dynamic Programming

Simon Wimmer Shuwei Hu Tobias Nipkow

Technical University of Munich

June 17, 2024

Abstract

We present a lightweight framework for the automatic verified (functional or imperative) memoization of recursive functions. Our tool can turn a pure Isabelle/HOL function definition into a monadified version in a state monad or the Imperative HOL heap monad, and prove a correspondence theorem. We provide a variety of memory implementations for the two types of monads. A number of simple techniques allow us to achieve bottom-up computation and space-efficient memoization. The framework's utility is demonstrated on a number of representative dynamic programming problems. A detailed description of our work can be found in the accompanying paper [2].

Contents

0.1 State Monad

theory *State_Monad_Ext*
imports *HOL−Library.State_Monad*
begin

definition *fun_app_lifted* :: ('*M*,'*a* ⇒ ('*M*, '*b*) *state*) *state* ⇒ ('*M*,'*a*) *state* ⇒ ('*M*,'*b*) *state* **where**
fun_app_lifted f_T x_T ≡ *do* { $f \leftarrow f_T$; $x \leftarrow x_T$; *f x* }

bundle *state_monad_syntax* **begin**

notation *fun_app_lifted* (**infixl** . 999)
type_synonym ('*a*,'*M*,'*b*) *fun_lifted* = '*a* ⇒ ('*M*,'*b*) *state* (_ ==_⟹ _ [3,1000,2] 2)
type_synonym ('*a*,'*b*) *dpfun* = '*a* ==('*a*⇀'*b*)⟹ '*b* (**infixr** ⇒$_T$ 2)
type_notation *state* ([_| _])

notation *State_Monad.return* (⟨_⟩)
notation (*ASCII*) *State_Monad.return* ((#_#))
notation *Transfer.Rel* (*Rel*)

end

context includes *state_monad_syntax* **begin**

qualified lemma *return_app_return*:
⟨*f*⟩ . ⟨*x*⟩ = *f x*
⟨*proof*⟩ **lemma** *return_app_return_meta*:
⟨*f*⟩ . ⟨*x*⟩ ≡ *f x*
⟨*proof*⟩ **definition** if_T :: ('*M*, *bool*) *state* ⇒ ('*M*, '*a*) *state* ⇒ ('*M*, '*a*) *state* ⇒ ('*M*, '*a*) *state* **where**
if_T b_T x_T y_T ≡ *do* {$b \leftarrow b_T$; *if b then* x_T *else* y_T}
end

end

1 Monadification

1.1 Monads

theory *Pure_Monad*
imports *Main*
begin

definition *Wrap* :: $'a \Rightarrow 'a$ **where**
Wrap $x \equiv x$

definition *App* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
App $f \equiv f$

lemma *Wrap_App_Wrap*:
App (*Wrap* f) (*Wrap* x) $\equiv f\ x$
⟨*proof*⟩

end

1.2 Parametricity of the State Monad

theory *DP_CRelVS*
imports *./State_Monad_Ext ../Pure_Monad*
begin

definition *lift_p* :: $('s \Rightarrow bool) \Rightarrow ('s, 'a)\ state \Rightarrow bool$ **where**
lift_p $P\ f =$
$(\forall\ heap.\ P\ heap \longrightarrow (case\ State\_Monad.run\_state\ f\ heap\ of\ (\_, heap) \Rightarrow P\ heap))$

context
fixes $P\ f\ heap$
assumes *lift*: *lift_p* $P\ f$ **and** P: $P\ heap$
begin

lemma *run_state_cases*:
case State_Monad.run_state $f\ heap\ of\ (\_, heap) \Rightarrow P\ heap$
⟨*proof*⟩

lemma *lift_p_P*:
$P\ heap'$ **if** *State_Monad.run_state* $f\ heap = (v, heap')$
⟨*proof*⟩

end

locale *state_mem_defs* =
fixes *lookup* :: $'param \Rightarrow ('mem, 'result\ option)\ state$
and *update* :: $'param \Rightarrow 'result \Rightarrow ('mem, unit)\ state$
begin

definition *checkmem* :: *'param* ⇒ (*'mem*, *'result*) *state* ⇒ (*'mem*, *'result*) *state* **where**
  checkmem param calc ≡ *do* {
    x ← *lookup param*;
    case x of
      Some x ⇒ *State_Monad.return x*
    | *None* ⇒ *do* {
        x ← *calc*;
        update param x;
        State_Monad.return x
      }
  }

abbreviation *checkmem_eq* ::
  (*'param* ⇒ (*'mem*, *'result*) *state*) ⇒ *'param* ⇒ (*'mem*, *'result*) *state* ⇒ *bool*
  (_$ _ =*CHECKMEM*= _ [1000,51] 51) **where**
  (*dp*$ *param* =*CHECKMEM*= *calc*) ≡ (*dp param* = *checkmem param calc*)
term 0

definition *map_of* **where**
  map_of heap k = *fst* (*run_state* (*lookup k*) *heap*)

definition *checkmem'* :: *'param* ⇒ (*unit* ⇒ (*'mem*, *'result*) *state*) ⇒ (*'mem*, *'result*) *state* **where**
  checkmem' param calc ≡ *do* {
    x ← *lookup param*;
    case x of
      Some x ⇒ *State_Monad.return x*
    | *None* ⇒ *do* {
        x ← *calc* ();
        update param x;
        State_Monad.return x
      }
  }

lemma *checkmem_checkmem'*:
  checkmem' param (λ_. *calc*) = *checkmem param calc*
  ⟨*proof*⟩

lemma *checkmem_eq_alt*:
  checkmem_eq dp param calc = (*dp param* = *checkmem' param* (λ _. *calc*))

⟨*proof*⟩

end

locale *mem_correct* = *state_mem_defs* +
fixes *P*
assumes *lookup_inv*: *lift_p P* (*lookup k*) **and** *update_inv*: *lift_p P* (*update k v*)
assumes
lookup_correct: *P m* ⟹ *map_of* (*snd* (*State_Monad.run_state* (*lookup k*) *m*)) ⊆$_m$ (*map_of m*)
and
update_correct: *P m* ⟹ *map_of* (*snd* (*State_Monad.run_state* (*update k v*) *m*)) ⊆$_m$ (*map_of m*)(*k* ↦ *v*)

locale *dp_consistency* =
mem_correct lookup update P
for *lookup* :: *'param* ⇒ (*'mem*, *'result option*) *state* **and** *update* **and** *P* +
fixes *dp* :: *'param* ⇒ *'result*
begin

context
includes *lifting_syntax state_monad_syntax*
begin

definition *cmem* :: *'mem* ⇒ *bool* **where**
cmem M ≡ ∀ *param*∈*dom* (*map_of M*). *map_of M param* = *Some* (*dp param*)

definition *crel_vs* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a* ⇒ (*'mem*, *'b*) *state* ⇒ *bool* **where**
crel_vs R v s ≡ ∀ *M*. *cmem M* ∧ *P M* ⟶ (*case State_Monad.run_state s M of* (*v'*, *M'*) ⇒ *R v v'* ∧ *cmem M'* ∧ *P M'*)

abbreviation *rel_fun_lifted* :: (*'a* ⇒ *'c* ⇒ *bool*) ⇒ (*'b* ⇒ *'d* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ (*'c* ==_⟹ *'d*) ⇒ *bool* (**infixr** ===>$_T$ 55) **where**
rel_fun_lifted R R' ≡ *R* ===> *crel_vs R'*
term 0

definition *consistentDP* :: (*'param* == *'mem* ⟹ *'result*) ⇒ *bool* **where**
consistentDP ≡ ((=) ===> *crel_vs* (=)) *dp*
term 0

private lemma *cmem_intro*:
assumes $\bigwedge$*param v M'. State_Monad.run_state* (*lookup param*) *M* = (*Some v*, *M'*) $\Longrightarrow$ *v* = *dp param*
shows *cmem M*
⟨*proof*⟩

lemma *cmem_elim*:
assumes *cmem M State_Monad.run_state* (*lookup param*) *M* = (*Some v*, *M'*)
obtains *dp param* = *v*
⟨*proof*⟩
term 0

lemma *crel_vs_intro*:
assumes $\bigwedge$*M v' M'*. ⟦*cmem M*; *P M*; *State_Monad.run_state* v_T *M* = (*v'*, *M'*)⟧ $\Longrightarrow$ *R v v'* $\wedge$ *cmem M'* $\wedge$ *P M'*
shows *crel_vs R v* v_T
⟨*proof*⟩
term 0

lemma *crel_vs_elim*:
assumes *crel_vs R v* v_T *cmem M P M*
obtains *v' M'* **where** *State_Monad.run_state* v_T *M* = (*v'*, *M'*) *R v v' cmem M' P M'*
⟨*proof*⟩
term 0

lemma *consistentDP_intro*:
assumes $\bigwedge$*param. Transfer.Rel* (*crel_vs* (=)) (*dp param*) (dp_T *param*)
shows *consistentDP* dp_T
⟨*proof*⟩

lemma *crel_vs_return*:
⟦*Transfer.Rel R x y*⟧ $\Longrightarrow$ *Transfer.Rel* (*crel_vs R*) (*Wrap x*) (*State_Monad.return y*)
⟨*proof*⟩
term 0

lemma *crel_vs_return_ext*:
⟦*Transfer.Rel R x y*⟧ $\Longrightarrow$ *Transfer.Rel* (*crel_vs R*) *x* (*State_Monad.return*

y)
⟨*proof*⟩
term 0

private lemma *cmem_upd*:
cmem M' **if** *cmem* M P M *State_Monad.run_state* (*update param* (*dp param*)) $M = (v, M')$
⟨*proof*⟩ **lemma** *P_upd*:
P M' **if** P M *State_Monad.run_state* (*update param* (*dp param*)) $M = (v, M')$
⟨*proof*⟩ **lemma** *crel_vs_get*:
$[\![\bigwedge M.\ cmem\ M \Longrightarrow crel\_vs\ R\ v\ (sf\ M)]\!] \Longrightarrow crel\_vs\ R\ v$ (*State_Monad.get* $\gg\!=$ *sf*)
⟨*proof*⟩
term 0

private lemma *crel_vs_set*:
$[\![crel\_vs\ R\ v\ sf;\ cmem\ M;\ P\ M]\!] \Longrightarrow crel\_vs\ R\ v$ (*State_Monad.set* M $\gg$ *sf*)
⟨*proof*⟩
term 0

private lemma *crel_vs_bind_eq*:
$[\![crel\_vs\ (=)\ v\ s;\ crel\_vs\ R\ (f\ v)\ (sf\ v)]\!] \Longrightarrow crel\_vs\ R\ (f\ v)\ (s \gg\!= sf)$
⟨*proof*⟩
term 0

lemma *bind_transfer*[*transfer_rule*]:
$(crel\_vs\ R0 ===> (R0 ===>_T R1) ===> crel\_vs\ R1)\ (\lambda v\ f.\ f\ v)\ (\gg\!=)$
⟨*proof*⟩ **lemma** *cmem_lookup*:
cmem M' **if** *cmem* M P M *State_Monad.run_state* (*lookup param*) $M = (v, M')$
⟨*proof*⟩ **lemma** *P_lookup*:
P M' **if** P M *State_Monad.run_state* (*lookup param*) $M = (v, M')$
⟨*proof*⟩

lemma *crel_vs_lookup*:
crel_vs (λ v v'. *case* v' *of None* $\Rightarrow$ *True* | *Some* $v' \Rightarrow v = v' \wedge v =$ *dp param*) (*dp param*) (*lookup param*)
⟨*proof*⟩

lemma *crel_vs_update*:
crel_vs (=) () (*update param* (*dp param*))

⟨*proof*⟩ **lemma** *crel_vs_checkmem*:
⟦*is_equality R*; *Transfer.Rel* (*crel_vs R*) (*dp param*) *s*⟧
⟹ *Transfer.Rel* (*crel_vs R*) (*dp param*) (*checkmem param s*)
⟨*proof*⟩

lemma *crel_vs_checkmem_tupled*:
assumes *v* = *dp param*
shows ⟦*is_equality R*; *Transfer.Rel* (*crel_vs R*) *v s*⟧
⟹ *Transfer.Rel* (*crel_vs R*) *v* (*checkmem param s*)
⟨*proof*⟩

lemma *return_transfer*[*transfer_rule*]:
(*R* ===>$_T$ *R*) *Wrap State_Monad.return*
⟨*proof*⟩

lemma *fun_app_lifted_transfer*[*transfer_rule*]:
(*crel_vs* (*R*0 ===>$_T$ *R*1) ===> *crel_vs R*0 ===> *crel_vs R*1) *App* (.)
⟨*proof*⟩

lemma *crel_vs_fun_app*:
⟦*Transfer.Rel* (*crel_vs R*0) *x* x_T; *Transfer.Rel* (*crel_vs* (*R*0 ===>$_T$ *R*1)) *f* f_T⟧ ⟹ *Transfer.Rel* (*crel_vs R*1) (*App f x*) (f_T . x_T)
⟨*proof*⟩

lemma if_T*_transfer*[*transfer_rule*]:
(*crel_vs* (=) ===> *crel_vs R* ===> *crel_vs R* ===> *crel_vs R*) *If State_Monad_Ext.*if_T
⟨*proof*⟩
end

end
end

1.3 Miscellaneous Parametricity Theorems

theory *State_Heap_Misc*
imports *Main*
begin
context includes *lifting_syntax* **begin**
lemma *rel_fun_comp*:
assumes (*R*1 ===> *S*1) *f g* (*R*2 ===> *S*2) *g h*

shows $(R1\ OO\ R2 ===> S1\ OO\ S2)\ f\ h$
⟨*proof*⟩

lemma *rel_fun_comp*1:
assumes $(R1 ===> S1)\ f\ g\ (R2 ===> S2)\ g\ h\ R' = R1\ OO\ R2$
shows $(R' ===> S1\ OO\ S2)\ f\ h$
⟨*proof*⟩

lemma *rel_fun_comp*2:
assumes $(R1 ===> S1)\ f\ g\ (R2 ===> S2)\ g\ h\ S' = S1\ OO\ S2$
shows $(R1\ OO\ R2 ===> S')\ f\ h$
⟨*proof*⟩

lemma *rel_fun_relcompp*:
$((R1 ===> S1)\ OO\ (R2 ===> S2))\ a\ b \Longrightarrow ((R1\ OO\ R2) ===> (S1\ OO\ S2))\ a\ b$
⟨*proof*⟩

lemma *rel_fun_comp*1′:
assumes $(R1 ===> S1)\ f\ g\ (R2 ===> S2)\ g\ h \bigwedge a\ b.\ R'\ a\ b \Longrightarrow (R1\ OO\ R2)\ a\ b$
shows $(R' ===> S1\ OO\ S2)\ f\ h$
⟨*proof*⟩

lemma *rel_fun_comp*2′:
assumes $(R1 ===> S1)\ f\ g\ (R2 ===> S2)\ g\ h \bigwedge a\ b.\ (S1\ OO\ S2)\ a\ b \Longrightarrow S'\ a\ b$
shows $(R1\ OO\ R2 ===> S')\ f\ h$
⟨*proof*⟩

end
end

1.4 Heap Monad

theory *Heap_Monad_Ext*
imports *HOL−Imperative_HOL.Imperative_HOL*
begin

definition *fun_app_lifted* :: $('a \Rightarrow 'b\ Heap)\ Heap \Rightarrow 'a\ Heap \Rightarrow 'b\ Heap$
where
$fun\_app\_lifted\ f_T\ x_T \equiv do\ \{\ f \leftarrow f_T;\ x \leftarrow x_T;\ f\ x\ \}$

bundle *heap_monad_syntax* **begin**

notation *fun_app_lifted* (**infixl** . 999)
type_synonym ('*a*, '*b*) *fun_lifted* = '*a* ⇒ '*b Heap* (_ ==*H*⟹ _ [3,2] 2)
type_notation *Heap* ([_])

notation *Heap_Monad.return* (⟨_⟩)
notation (*ASCII*) *Heap_Monad.return* ((#_#))
notation *Transfer.Rel* (*Rel*)

end

context includes *heap_monad_syntax* **begin**

qualified lemma *return_app_return*:
⟨*f*⟩ . ⟨*x*⟩ = *f x*
⟨*proof*⟩ **lemma** *return_app_return_meta*:
⟨*f*⟩ . ⟨*x*⟩ ≡ *f x*
⟨*proof*⟩ **definition** *if*$_T$ :: *bool Heap* ⇒ '*a Heap* ⇒ '*a Heap* ⇒ '*a Heap*
where
if$_T$ *b*$_T$ *x*$_T$ *y*$_T$ ≡ *do* {*b* ← *b*$_T$; *if b then x*$_T$ *else y*$_T$}
end

end

1.5 Relation Between the State and the Heap Monad

theory *State_Heap*
imports
../state_monad/DP_CRelVS
HOL−Imperative_HOL.Imperative_HOL
State_Heap_Misc
Heap_Monad_Ext
begin

definition *lift_p* :: (*heap* ⇒ *bool*) ⇒ '*a Heap* ⇒ *bool* **where**
lift_p P f =
(∀ *heap*. *P heap* ⟶ (*case execute f heap of None* ⇒ *False* | *Some* (_, *heap*) ⇒ *P heap*))

context
fixes *P f heap*
assumes *lift*: *lift_p P f* **and** *P*: *P heap*
begin

lemma *execute_cases*:
case execute f heap of None ⇒ False | Some (_, heap) ⇒ P heap
⟨*proof*⟩

lemma *execute_cases′*:
case execute f heap of Some (_, heap) ⇒ P heap
⟨*proof*⟩

lemma *lift_p_None*[*simp*, *dest*]:
False **if** *execute f heap = None*
⟨*proof*⟩

lemma *lift_p_P*:
case the (*execute f heap*) *of* (_, *heap*) ⇒ *P heap*
⟨*proof*⟩

lemma *lift_p_P′*:
P heap′ **if** *the* (*execute f heap*) = (*v*, *heap′*)
⟨*proof*⟩

lemma *lift_p_P″*:
P heap′ **if** *execute f heap = Some* (*v*, *heap′*)
⟨*proof*⟩

lemma *lift_p_the_Some*[*simp*]:
execute f heap = Some (*v*, *heap′*) **if** *the* (*execute f heap*) = (*v*, *heap′*)
⟨*proof*⟩

lemma *lift_p_E*:
obtains *v heap′* **where** *execute f heap = Some* (*v*, *heap′*) *P heap′*
⟨*proof*⟩

end

definition *state_of s* ≡ *State* (λ *heap*. *the* (*execute s heap*))

locale *heap_mem_defs* =
fixes *P* :: *heap* ⇒ *bool*
and *lookup* :: *′k* ⇒ *′v option Heap*
and *update* :: *′k* ⇒ *′v* ⇒ *unit Heap*
begin

definition *rel_state* :: (*′a* ⇒ *′b* ⇒ *bool*) ⇒ (*heap*, *′a*) *state* ⇒ *′b Heap* ⇒ *bool* **where**

```
  rel_state R f g ≡
    ∀ heap. P heap ⟶
      (case State_Monad.run_state f heap of (v1, heap1) ⇒ case execute g
heap of
        Some (v2, heap2) ⇒ R v1 v2 ∧ heap1 = heap2 ∧ P heap2 | None ⇒
False)

definition lookup' k ≡ State (λ heap. the (execute (lookup k) heap))

definition update' k v ≡ State (λ heap. the (execute (update k v) heap))

definition heap_get = Heap_Monad.Heap (λ heap. Some (heap, heap))

definition checkmem :: 'k ⇒ 'v Heap ⇒ 'v Heap where
  checkmem param calc ≡
    Heap_Monad.bind (lookup param) (λ x.
    case x of
      Some x ⇒ return x
    | None ⇒ Heap_Monad.bind calc (λ x.
        Heap_Monad.bind (update param x) (λ _.
        return x
      )
    )
  )


definition checkmem' :: 'k ⇒ (unit ⇒ 'v Heap) ⇒ 'v Heap where
  checkmem' param calc ≡
    Heap_Monad.bind (lookup param) (λ x.
    case x of
      Some x ⇒ return x
    | None ⇒ Heap_Monad.bind (calc ()) (λ x.
        Heap_Monad.bind (update param x) (λ _.
        return x
      )
    )
  )


lemma checkmem_checkmem':
  checkmem' param (λ_. calc) = checkmem param calc
  ⟨proof⟩

definition map_of_heap where
```

map_of_heap heap k = fst (the (execute (lookup k) heap))

lemma *rel_state_elim*:
assumes *rel_state R f g P heap*
obtains *heap' v v'* **where**
State_Monad.run_state f heap = (v, heap') execute g heap = Some (v', heap') R v v' P heap'
⟨*proof*⟩

lemma *rel_state_intro*:
assumes
$\bigwedge$ *heap v heap'. P heap* $\Longrightarrow$ *State_Monad.run_state f heap = (v, heap')* $\Longrightarrow$ $\exists$ *v'. R v v'* $\wedge$ *execute g heap = Some (v', heap')*
$\bigwedge$ *heap v heap'. P heap* $\Longrightarrow$ *State_Monad.run_state f heap = (v, heap')* $\Longrightarrow$ *P heap'*
shows *rel_state R f g*
⟨*proof*⟩

context
includes *lifting_syntax state_monad_syntax*
begin

lemma *transfer_bind*[*transfer_rule*]:
(rel_state R ===> (R ===> rel_state Q) ===> rel_state Q) State_Monad.bind Heap_Monad.bind
⟨*proof*⟩

lemma *transfer_return*[*transfer_rule*]:
(R ===> rel_state R) State_Monad.return Heap_Monad.return
⟨*proof*⟩

lemma *fun_app_lifted_transfer*:
(rel_state (R ===> rel_state Q) ===> rel_state R ===> rel_state Q)
State_Monad_Ext.fun_app_lifted Heap_Monad_Ext.fun_app_lifted
⟨*proof*⟩

lemma *transfer_get*[*transfer_rule*]:
rel_state (=) State_Monad.get heap_get
⟨*proof*⟩

end

end

locale *heap_inv* = *heap_mem_defs* _ *lookup* **for** *lookup* :: *'k* ⇒ *'v option*
Heap +
  assumes *lookup_inv*: *lift_p P* (*lookup k*)
  assumes *update_inv*: *lift_p P* (*update k v*)
begin

lemma *rel_state_lookup*:
  rel_state (=) (*lookup' k*) (*lookup k*)
  ⟨*proof*⟩

lemma *rel_state_update*:
  rel_state (=) (*update' k v*) (*update k v*)
  ⟨*proof*⟩

context
  includes *lifting_syntax*
begin

lemma *transfer_lookup*:
  ((=) ===> *rel_state* (=)) *lookup' lookup*
  ⟨*proof*⟩

lemma *transfer_update*:
  ((=) ===> (=) ===> *rel_state* (=)) *update' update*
  ⟨*proof*⟩

lemma *transfer_checkmem*:
  ((=) ===> *rel_state* (=) ===> *rel_state* (=))
    (*state_mem_defs.checkmem lookup' update'*) *checkmem*
  ⟨*proof*⟩

end

end

locale *heap_correct* =
  heap_inv +
  assumes *lookup_correct*:
        P m ⟹ *map_of_heap* (*snd* (*the* (*execute* (*lookup k*) *m*))) ⊆$_m$
(*map_of_heap m*)
  and *update_correct*:
        P m ⟹ *map_of_heap* (*snd* (*the* (*execute* (*update k v*) *m*))) ⊆$_m$
(*map_of_heap m*)(*k* ↦ *v*)

begin

lemma *lookup′_correct*:
state_mem_defs.map_of lookup′ (*snd* (*State_Monad.run_state* (*lookup′ k*) *m*)) ⊆$_m$ (*state_mem_defs.map_of lookup′ m*) **if** *P m*
⟨*proof*⟩

lemma *update′_correct*:
state_mem_defs.map_of lookup′ (*snd* (*State_Monad.run_state* (*update′ k v*) *m*)) ⊆$_m$ (*state_mem_defs.map_of lookup′ m*)(*k* ↦ *v*)
if *P m*
⟨*proof*⟩

lemma *lookup′_inv*:
DP_CRelVS.lift_p P (*lookup′ k*)
⟨*proof*⟩

lemma *update′_inv*:
DP_CRelVS.lift_p P (*update′ k v*)
⟨*proof*⟩

lemma *mem_correct_heap*: *mem_correct lookup′ update′ P*
⟨*proof*⟩

end

context *heap_mem_defs*
begin

context
includes *lifting_syntax*
begin

lemma *mem_correct_heap_correct*:
assumes *correct*: *mem_correct lookup*$_s$ *update*$_s$ *P*
and *lookup*: ((=) ===> *rel_state* (=)) *lookup*$_s$ *lookup*
and *update*: ((=) ===> (=) ===> *rel_state* (=)) *update*$_s$ *update*
shows *heap_correct P update lookup*
⟨*proof*⟩

end

end

end

1.6 Parametricity of the Heap Monad

theory *DP_CRelVH*
imports *State_Heap*
begin

locale *dp_heap* =
state_dp_consistency: *dp_consistency lookup_st update_st P dp* + *heap_mem_defs Q lookup update*
for *P Q* :: *heap* ⇒ *bool* **and** *dp* :: *'k* ⇒ *'v* **and** *lookup* :: *'k* ⇒ *'v option Heap*
and *lookup_st update update_st* +
assumes
rel_state_lookup: *rel_fun* (=) (*rel_state* (=)) *lookup_st lookup*
and
rel_state_update: *rel_fun* (=) (*rel_fun* (=) (*rel_state* (=))) *update_st update*
begin

context
includes *lifting_syntax heap_monad_syntax*
begin

definition *crel_vs R v f* ≡
∀ *heap*. *P heap* ∧ *Q heap* ∧ *state_dp_consistency.cmem heap* ⟶
(*case execute f heap of*
None ⇒ *False* |
Some (*v'*, *heap'*) ⇒ *P heap'* ∧ *Q heap'* ∧ *R v v'* ∧ *state_dp_consistency.cmem heap'*
)

abbreviation *rel_fun_lifted* :: (*'a* ⇒ *'c* ⇒ *bool*) ⇒ (*'b* ⇒ *'d* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ (*'c* ==*H*⟹ *'d*) ⇒ *bool* (**infixr** ===>$_T$ 55) **where**
rel_fun_lifted R R' ≡ *R* ===> *crel_vs R'*

definition *consistentDP* :: (*'k* ⇒ *'v Heap*) ⇒ *bool* **where**
consistentDP ≡ ((=) ===> *crel_vs* (=)) *dp*

lemma *consistentDP_intro*:
assumes ⋀*param*. *Transfer.Rel* (*crel_vs* (=)) (*dp param*) (*dp*$_T$ *param*)

shows *consistentDP* dp_T
⟨*proof*⟩

lemma *crel_vs_execute_None*:
False **if** *crel_vs R a b execute b heap = None P heap Q heap state_dp_consistency.cmem heap*
⟨*proof*⟩

lemma *crel_vs_execute_Some*:
assumes *crel_vs R a b P heap Q heap state_dp_consistency.cmem heap*
obtains *x heap'* **where** *execute b heap = Some (x, heap') P heap' Q heap'*
⟨*proof*⟩

lemma *crel_vs_executeD*:
assumes *crel_vs R a b P heap Q heap state_dp_consistency.cmem heap*
obtains *x heap'* **where**
execute b heap = Some (x, heap') P heap' Q heap' state_dp_consistency.cmem heap' R a x
⟨*proof*⟩

lemma *crel_vs_success*:
assumes *crel_vs R a b P heap Q heap state_dp_consistency.cmem heap*
shows *success b heap*
⟨*proof*⟩

lemma *crel_vsI*: *crel_vs R a b* **if** *(state_dp_consistency.crel_vs R OO rel_state (=)) a b*
⟨*proof*⟩

lemma *transfer'_return*[*transfer_rule*]:
(R ===> crel_vs R) Wrap return
⟨*proof*⟩

lemma *crel_vs_return*:
Transfer.Rel (crel_vs R) (Wrap x) (return y) **if** *Transfer.Rel R x y*
⟨*proof*⟩

lemma *crel_vs_return_ext*:
⟦*Transfer.Rel R x y*⟧ ⟹ *Transfer.Rel (crel_vs R) x (Heap_Monad.return y)*
⟨*proof*⟩
term 0

lemma *bind_transfer*[*transfer_rule*]:

(*crel_vs R0 ===> (R0 ===> crel_vs R1) ===> crel_vs R1*) (λ*v f*. *f v*) (≫=)
⟨*proof*⟩

lemma *crel_vs_update*:
crel_vs (=) () (*update param* (*dp param*))
⟨*proof*⟩

lemma *crel_vs_lookup*:
crel_vs
(λ *v v'*. *case v' of None* ⇒ *True* | *Some v'* ⇒ *v* = *v'* ∧ *v* = *dp param*)
(*dp param*) (*lookup param*)
⟨*proof*⟩

lemma *crel_vs_eq_eq_onp*:
crel_vs (*eq_onp* (λ *x*. *x* = *v*)) *v s* **if** *crel_vs* (=) *v s*
⟨*proof*⟩

lemma *crel_vs_bind_eq*:
⟦*crel_vs* (=) *v s*; *crel_vs R* (*f v*) (*sf v*)⟧ ⟹ *crel_vs R* (*f v*) (*s* ≫= *sf*)
⟨*proof*⟩

lemma *crel_vs_checkmem*:
Transfer.Rel (*crel_vs R*) (*dp param*) (*checkmem param s*) **if** *is_equality R Transfer.Rel* (*crel_vs R*) (*dp param*) *s*
⟨*proof*⟩

lemma *crel_vs_checkmem_tupled*:
assumes *v* = *dp param*
shows ⟦*is_equality R*; *Transfer.Rel* (*crel_vs R*) *v s*⟧
⟹ *Transfer.Rel* (*crel_vs R*) *v* (*checkmem param s*)
⟨*proof*⟩

lemma *transfer_fun_app_lifted*[*transfer_rule*]:
(*crel_vs* (*R0* ===> *crel_vs R1*) ===> *crel_vs R0* ===> *crel_vs R1*)
App Heap_Monad_Ext.fun_app_lifted
⟨*proof*⟩

lemma *crel_vs_fun_app*:
⟦*Transfer.Rel* (*crel_vs R0*) *x* x_T; *Transfer.Rel* (*crel_vs* (*R0* ===>$_T$ *R1*))
f f_T⟧ ⟹ *Transfer.Rel* (*crel_vs R1*) (*App f x*) (f_T . x_T)
⟨*proof*⟩

end

end

locale *dp_consistency_heap* = *heap_correct* +
  fixes *dp* :: *'a* ⇒ *'b*
begin

interpretation *state_mem_correct*: *mem_correct lookup' update' P*
  ⟨*proof*⟩

interpretation *state_dp_consistency*: *dp_consistency lookup' update' P dp*
⟨*proof*⟩

lemma *dp_heap*: *dp_heap P P lookup lookup' update update'*
  ⟨*proof*⟩

sublocale *dp_heap P P dp lookup lookup' update update'*
  ⟨*proof*⟩

notation *rel_fun_lifted* (**infixr** ===>$_T$ 55)
end

locale *heap_correct_empty* = *heap_correct* +
  fixes *empty*
 assumes *empty_correct*: *map_of_heap empty* ⊆$_m$ *Map.empty* **and** *P_empty*:
P empty

locale *dp_consistency_heap_empty* =
  dp_consistency_heap + *heap_correct_empty*
begin

lemma *cmem_empty*:
  state_dp_consistency.cmem empty
  ⟨*proof*⟩

corollary *memoization_correct*:
  dp x = *v state_dp_consistency.cmem m* **if**
  consistentDP dp$_T$ *Heap_Monad.execute* (*dp*$_T$ *x*) *empty* = *Some* (*v*, *m*)
  ⟨*proof*⟩

lemma *memoized_success*:
  success (*dp*$_T$ *x*) *empty* **if** *consistentDP dp*$_T$
  ⟨*proof*⟩

lemma *memoized*:
dp x = fst (the (Heap_Monad.execute (dp$_T$ x) empty)) **if** *consistentDP dp$_T$*
⟨*proof*⟩

lemma *cmem_result*:
state_dp_consistency.cmem (snd (the (Heap_Monad.execute (dp$_T$ x) empty))) **if** *consistentDP dp$_T$*
⟨*proof*⟩

end

end

2 Memoization

2.1 Memory Implementations for the State Monad

theory *Memory*
imports *DP_CRelVS HOL−Library.Mapping*
begin

lemma *lift_pI*[*intro?*]:
lift_p P f **if** ⋀ *heap x heap′. P heap ⟹ run_state f heap = (x, heap′) ⟹ P heap′*
⟨*proof*⟩

lemma *mem_correct_default*:
mem_correct
(λ k. do {m ← State_Monad.get; State_Monad.return (m k)})
(λ k v. do {m ← State_Monad.get; State_Monad.set (m(k↦v))})
(λ _. True)
⟨*proof*⟩

lemma *mem_correct_rbt_mapping*:
mem_correct
(λ k. do {m ← State_Monad.get; State_Monad.return (Mapping.lookup m k)})
(λ k v. do {m ← State_Monad.get; State_Monad.set (Mapping.update k v m)})
(λ _. True)
⟨*proof*⟩

locale *mem_correct_empty* = *mem_correct* +
  fixes *empty*
  assumes *empty_correct*: *map_of empty* $\subseteq_m$ *Map.empty* **and** *P_empty*: *P empty*

lemma (**in** *mem_correct_empty*) *dom_empty*[*simp*]:
  dom (*map_of empty*) = {}
  ⟨*proof*⟩

lemma *mem_correct_empty_default*:
  mem_correct_empty
    (λ *k*. *do* {*m* ← *State_Monad.get*; *State_Monad.return* (*m k*)})
    (λ *k v*. *do* {*m* ← *State_Monad.get*; *State_Monad.set* (*m*(*k*↦*v*))})
    (λ _. *True*)
    Map.empty
  ⟨*proof*⟩

lemma *mem_correct_rbt_empty_mapping*:
  mem_correct_empty
    (λ *k*. *do* {*m* ← *State_Monad.get*; *State_Monad.return* (*Mapping.lookup m k*)})
    (λ *k v*. *do* {*m* ← *State_Monad.get*; *State_Monad.set* (*Mapping.update k v m*)})
    (λ _. *True*)
    Mapping.empty
  ⟨*proof*⟩

locale *dp_consistency_empty* =
  dp_consistency + *mem_correct_empty*
begin

lemma *cmem_empty*:
  cmem empty
  ⟨*proof*⟩

corollary *memoization_correct*:
  dp x = *v cmem m* **if** *consistentDP dp*$_T$ *State_Monad.run_state* (*dp*$_T$ *x*) *empty* = (*v*, *m*)
  ⟨*proof*⟩

lemma *memoized*:

dp x = fst (State_Monad.run_state (dp$_T$ x) empty) **if** *consistentDP dp$_T$*
⟨*proof*⟩

lemma *cmem_result*:
cmem (snd (State_Monad.run_state (dp$_T$ x) empty)) **if** *consistentDP dp$_T$*
⟨*proof*⟩

end

locale *dp_consistency_default* =
fixes *dp* :: *'param* ⇒ *'result*
begin

sublocale *dp_consistency_empty*
λ *k*. *do* {(*m*::*'param* ⇀ *'result*) ← *State_Monad.get*; *State_Monad.return* (*m k*)}
λ *k v*. *do* {*m* ← *State_Monad.get*; *State_Monad.set* (*m*(*k*↦*v*))}
λ (_::*'param* ⇀ *'result*). *True*
dp
Map.empty
⟨*proof*⟩

end

locale *dp_consistency_mapping* =
fixes *dp* :: *'param* ⇒ *'result*
begin

sublocale *dp_consistency_empty*
(λ *k*. *do* {(*m*::(*'param*,*'result*) *mapping*) ← *State_Monad.get*; *State_Monad.return* (*Mapping.lookup m k*)})
(λ *k v*. *do* {*m* ← *State_Monad.get*; *State_Monad.set* (*Mapping.update k v m*)})
(λ _::(*'param*,*'result*) *mapping*. *True*) *dp Mapping.empty*
⟨*proof*⟩

end

2.1.1 Tracing Memory

context *state_mem_defs*
begin

definition

lookup_trace k =
State (λ (log, m). case State_Monad.run_state (lookup k) m of
(None, m) ⇒ (None, (''Missed'', k) # log, m)) |
(Some v, m) ⇒ (Some v, (''Found'', k) # log, m))
)

definition
update_trace k v =
State (λ (log, m). case State_Monad.run_state (update k v) m of
(_, m) ⇒ ((), (''Stored'', k) # log, m))
)

end

context *mem_correct*
begin

lemma *map_of_simp*:
state_mem_defs.map_of lookup_trace = map_of o snd
⟨*proof*⟩

lemma *mem_correct_tracing*: *mem_correct lookup_trace update_trace (P o snd)*
⟨*proof*⟩

end

context *mem_correct_empty*
begin

lemma *mem_correct_tracing_empty*:
mem_correct_empty lookup_trace update_trace (P o snd) ([], empty)
⟨*proof*⟩

end

locale *dp_consistency_mapping_tracing =*
fixes *dp* :: *'param ⇒ 'result*
begin

interpretation *mapping*: *dp_consistency_mapping* ⟨*proof*⟩

sublocale *dp_consistency_empty*
mapping.lookup_trace mapping.update_trace (λ _. True) o snd dp ([],

Mapping.empty)
⟨*proof*⟩

end

end

2.2 Pair Memory

theory *Pair_Memory*
imports *../state_monad/Memory*
begin

lemma *map_add_mono*:
$(m1 +\!\!+ m2) \subseteq_m (m1' +\!\!+ m2')$ **if** $m1 \subseteq_m m1'$ $m2 \subseteq_m m2'$ *dom* $m1 \cap$ *dom* $m2' = \{\}$
⟨*proof*⟩

lemma *map_add_upd*2:
$f(x \mapsto y) +\!\!+ g = (f +\!\!+ g)(x \mapsto y)$ **if** *dom* $f \cap$ *dom* $g = \{\}$ $x \notin$ *dom* g
⟨*proof*⟩

locale *pair_mem_defs* =
fixes *lookup*1 *lookup*2 :: $'a \Rightarrow ('mem, 'v\ option)\ state$
and *update*1 *update*2 :: $'a \Rightarrow 'v \Rightarrow ('mem, unit)\ state$
and *move*12 :: $'k1 \Rightarrow ('mem, unit)\ state$
and *get_k*1 *get_k*2 :: $('mem, 'k1)\ state$
and P :: $'mem \Rightarrow bool$
fixes *key*1 :: $'k \Rightarrow 'k1$ **and** *key*2 :: $'k \Rightarrow 'a$
begin

We assume that look-ups happen on the older row, so it is biased towards the second entry.

definition
lookup_pair k = *do* {
let k' = *key*1 k;
*k*2 ← *get_k*2;
if k' = *k*2
*then lookup*2 (*key*2 k)
else do {
*k*1 ← *get_k*1;
if k' = *k*1
*then lookup*1 (*key*2 k)

```
    else State_Monad.return None
  }
 }
```

We assume that updates happen on the newer row, so it is biased towards the first entry.

definition

```
update_pair k v = do {
  let k' = key1 k;
  k1 ← get_k1;
  if k' = k1
  then update1 (key2 k) v
  else do {
    k2 ← get_k2;
    if k' = k2
    then update2 (key2 k) v
    else (move12 k' ≫ update1 (key2 k) v)
  }
}
```

sublocale *pair*: *state_mem_defs lookup_pair update_pair* ⟨*proof*⟩

sublocale *mem*1: *state_mem_defs lookup*1 *update*1 ⟨*proof*⟩

sublocale *mem*2: *state_mem_defs lookup*2 *update*2 ⟨*proof*⟩

definition

```
inv_pair heap ≡
  let
    k1 = fst (State_Monad.run_state get_k1 heap);
    k2 = fst (State_Monad.run_state get_k2 heap)
  in
  (∀ k ∈ dom (mem1.map_of heap). ∃ k'. key1 k' = k1 ∧ key2 k' = k) ∧
  (∀ k ∈ dom (mem2.map_of heap). ∃ k'. key1 k' = k2 ∧ key2 k' = k) ∧
  k1 ≠ k2 ∧ P heap
```

definition

*map_of*1 *m k* = (*if key*1 *k* = *fst* (*State_Monad.run_state get_k*1 *m*) *then mem*1.*map_of m* (*key*2 *k*) *else None*)

definition

map_of2 m k = (if key1 k = fst (State_Monad.run_state get_k2 m) then mem2.map_of m (key2 k) else None)

end

locale *pair_mem = pair_mem_defs +*
assumes *get_state*:
State_Monad.run_state get_k1 m = (k, m') ⟹ m' = m
State_Monad.run_state get_k2 m = (k, m') ⟹ m' = m
assumes *move12_correct*:
P m ⟹ State_Monad.run_state (move12 k1) m = (x, m') ⟹ mem1.map_of m' ⊆m Map.empty
P m ⟹ State_Monad.run_state (move12 k1) m = (x, m') ⟹ mem2.map_of m' ⊆m mem1.map_of m
assumes *move12_keys*:
State_Monad.run_state (move12 k1) m = (x, m') ⟹ fst (State_Monad.run_state get_k1 m') = k1
State_Monad.run_state (move12 k1) m = (x, m') ⟹ fst (State_Monad.run_state get_k2 m') = fst (State_Monad.run_state get_k1 m)
assumes *move12_inv*:
lift_p P (move12 k1)
assumes *lookup_inv*:
lift_p P (lookup1 k') lift_p P (lookup2 k')
assumes *update_inv*:
lift_p P (update1 k' v) lift_p P (update2 k' v)
assumes *lookup_keys*:
P m ⟹ State_Monad.run_state (lookup1 k') m = (v', m') ⟹
fst (State_Monad.run_state get_k1 m') = fst (State_Monad.run_state get_k1 m)
P m ⟹ State_Monad.run_state (lookup1 k') m = (v', m') ⟹
fst (State_Monad.run_state get_k2 m') = fst (State_Monad.run_state get_k2 m)
P m ⟹ State_Monad.run_state (lookup2 k') m = (v', m') ⟹
fst (State_Monad.run_state get_k1 m') = fst (State_Monad.run_state get_k1 m)
P m ⟹ State_Monad.run_state (lookup2 k') m = (v', m') ⟹
fst (State_Monad.run_state get_k2 m') = fst (State_Monad.run_state get_k2 m)
assumes *update_keys*:
P m ⟹ State_Monad.run_state (update1 k' v) m = (x, m') ⟹
fst (State_Monad.run_state get_k1 m') = fst (State_Monad.run_state get_k1 m)
P m ⟹ State_Monad.run_state (update1 k' v) m = (x, m') ⟹
fst (State_Monad.run_state get_k2 m') = fst (State_Monad.run_state

get_k2 m)
  P m $\Longrightarrow$ *State_Monad.run_state* (*update2 k' v*) *m* = (*x*, *m'*) $\Longrightarrow$
  fst (*State_Monad.run_state get_k1 m'*) = *fst* (*State_Monad.run_state get_k1 m*)
  P m $\Longrightarrow$ *State_Monad.run_state* (*update2 k' v*) *m* = (*x*, *m'*) $\Longrightarrow$
  fst (*State_Monad.run_state get_k2 m'*) = *fst* (*State_Monad.run_state get_k2 m*)
 assumes
  lookup_correct:
   P m $\Longrightarrow$ *mem1.map_of* (*snd* (*State_Monad.run_state* (*lookup1 k'*) *m*)) $\subseteq_m$ (*mem1.map_of m*)
   P m $\Longrightarrow$ *mem2.map_of* (*snd* (*State_Monad.run_state* (*lookup1 k'*) *m*)) $\subseteq_m$ (*mem2.map_of m*)
   P m $\Longrightarrow$ *mem1.map_of* (*snd* (*State_Monad.run_state* (*lookup2 k'*) *m*)) $\subseteq_m$ (*mem1.map_of m*)
   P m $\Longrightarrow$ *mem2.map_of* (*snd* (*State_Monad.run_state* (*lookup2 k'*) *m*)) $\subseteq_m$ (*mem2.map_of m*)
 assumes
  update_correct:
   P m $\Longrightarrow$ *mem1.map_of* (*snd* (*State_Monad.run_state* (*update1 k' v*) *m*)) $\subseteq_m$ (*mem1.map_of m*)($k' \mapsto v$)
   P m $\Longrightarrow$ *mem2.map_of* (*snd* (*State_Monad.run_state* (*update2 k' v*) *m*)) $\subseteq_m$ (*mem2.map_of m*)($k' \mapsto v$)
   P m $\Longrightarrow$ *mem2.map_of* (*snd* (*State_Monad.run_state* (*update1 k' v*) *m*)) $\subseteq_m$ *mem2.map_of m*
   P m $\Longrightarrow$ *mem1.map_of* (*snd* (*State_Monad.run_state* (*update2 k' v*) *m*)) $\subseteq_m$ *mem1.map_of m*
begin

lemma *map_of_le_pair*:
 pair.map_of m $\subseteq_m$ *map_of1 m* ++ *map_of2 m*
 if *inv_pair m*
 ⟨*proof*⟩

lemma *pair_le_map_of*:
 map_of1 m ++ *map_of2 m* $\subseteq_m$ *pair.map_of m*
 if *inv_pair m*
 ⟨*proof*⟩

lemma *map_of_eq_pair*:
 map_of1 m ++ *map_of2 m* = *pair.map_of m*
 if *inv_pair m*
 ⟨*proof*⟩

lemma *inv_pair_neq*[*simp*]:
False **if** *inv_pair m fst* (*State_Monad.run_state get_k*1 *m*) = *fst* (*State_Monad.run_state get_k*2 *m*)
⟨*proof*⟩

lemma *inv_pair_P_D*:
P m **if** *inv_pair m*
⟨*proof*⟩

lemma *inv_pair_domD*[*intro*]:
dom (*map_of*1 *m*) ∩ *dom* (*map_of*2 *m*) = {} **if** *inv_pair m*
⟨*proof*⟩

lemma *move*12*_correct*1:
*map_of*1 *heap′* ⊆$_m$ *Map.empty* **if** *State_Monad.run_state* (*move*12 *k*1) *heap* = (*x*, *heap′*) *P heap*
⟨*proof*⟩

lemma *move*12*_correct*2:
*map_of*2 *heap′* ⊆$_m$ *map_of*1 *heap* **if** *State_Monad.run_state* (*move*12 *k*1) *heap* = (*x*, *heap′*) *P heap*
⟨*proof*⟩

lemma *dom_empty*[*simp*]:
dom (*map_of*1 *heap′*) = {} **if** *State_Monad.run_state* (*move*12 *k*1) *heap* = (*x*, *heap′*) *P heap*
⟨*proof*⟩

lemma *inv_pair_lookup*1:
inv_pair m′ **if** *State_Monad.run_state* (*lookup*1 *k*) *m* = (*v*, *m′*) *inv_pair m*
⟨*proof*⟩

lemma *inv_pair_lookup*2:
inv_pair m′ **if** *State_Monad.run_state* (*lookup*2 *k*) *m* = (*v*, *m′*) *inv_pair m*
⟨*proof*⟩

lemma *inv_pair_update*1:
inv_pair m′
if *State_Monad.run_state* (*update*1 (*key*2 *k*) *v*) *m* = (*v′*, *m′*) *inv_pair m* *fst* (*State_Monad.run_state get_k*1 *m*) = *key*1 *k*
⟨*proof*⟩

lemma *inv_pair_update2*:
inv_pair m'
if *State_Monad.run_state* (*update2* (*key2* *k*) *v*) *m* = (*v'*, *m'*) *inv_pair m*
fst (*State_Monad.run_state get_k2 m*) = *key1 k*
⟨*proof*⟩

lemma *inv_pair_move12*:
inv_pair m'
if *State_Monad.run_state* (*move12 k*) *m* = (*v'*, *m'*) *inv_pair m fst* (*State_Monad.run_state*
get_k1 m) ≠ *k*
⟨*proof*⟩

lemma *mem_correct_pair*:
mem_correct lookup_pair update_pair inv_pair
if *injective*: ∀ *k k'*. *key1 k* = *key1 k'* ∧ *key2 k* = *key2 k'* ⟶ *k* = *k'*
⟨*proof*⟩

lemma *emptyI*:
assumes *inv_pair m mem1.map_of m* ⊆$_m$ *Map.empty mem2.map_of m*
⊆$_m$ *Map.empty*
shows *pair.map_of m* ⊆$_m$ *Map.empty*
⟨*proof*⟩

end

datatype (*'k*, *'v*) *pair_storage* = *Pair_Storage 'k 'k 'v 'v*

context *mem_correct_empty*
begin

context
fixes *key* :: *'a* ⇒ *'k*
begin

We assume that look-ups happen on the older row, so it is biased towards the second entry.

definition
lookup_pair k =
State (λ *mem*.
(
case mem of Pair_Storage k1 k2 m1 m2 ⇒ *let k'* = *key k in*
if k' = *k2 then case State_Monad.run_state* (*lookup k*) *m2 of* (*v*, *m*)
⇒ (*v*, *Pair_Storage k1 k2 m1 m*)

else if $k' = k1$ *then case State_Monad.run_state* (*lookup* k) $m1$ *of* (v, m) ⇒ (v, *Pair_Storage* $k1$ $k2$ m $m2$)
else (*None*, *mem*)
)
)

We assume that updates happen on the newer row, so it is biased towards the first entry.

definition
update_pair k v =
State (λ *mem*.
(
case mem of Pair_Storage $k1$ $k2$ $m1$ $m2$ ⇒ *let* k' = *key* k *in*
if $k' = k1$ *then case State_Monad.run_state* (*update* k v) $m1$ *of* (_, m) ⇒ ((), *Pair_Storage* $k1$ $k2$ m $m2$)
else if $k' = k2$ *then case State_Monad.run_state* (*update* k v) $m2$ *of* (_, m) ⇒ ((),*Pair_Storage* $k1$ $k2$ $m1$ m)
else case State_Monad.run_state (*update* k v) *empty of* (_, m) ⇒ ((), *Pair_Storage* k' $k1$ m $m1$)
)
)

interpretation *pair*: *state_mem_defs lookup_pair update_pair* ⟨*proof*⟩

definition
inv_pair p = (*case* p *of Pair_Storage* $k1$ $k2$ $m1$ $m2$ ⇒
key ‘ *dom* (*map_of* $m1$) ⊆ {$k1$} ∧ *key* ‘ *dom* (*map_of* $m2$) ⊆ {$k2$} ∧ $k1$ ≠ $k2$ ∧ P $m1$ ∧ P $m2$
)

lemma *map_of_le_pair*:
pair.map_of (*Pair_Storage* $k1$ $k2$ $m1$ $m2$) ⊆$_m$ (*map_of* $m1$ ++ *map_of* $m2$)
if *inv_pair* (*Pair_Storage* $k1$ $k2$ $m1$ $m2$)
⟨*proof*⟩

lemma *pair_le_map_of*:
map_of $m1$ ++ *map_of* $m2$ ⊆$_m$ *pair.map_of* (*Pair_Storage* $k1$ $k2$ $m1$ $m2$)
if *inv_pair* (*Pair_Storage* $k1$ $k2$ $m1$ $m2$)
⟨*proof*⟩

lemma *map_of_eq_pair*:
map_of m1 ++ map_of m2 = pair.map_of (Pair_Storage k1 k2 m1 m2)
if *inv_pair (Pair_Storage k1 k2 m1 m2)*
⟨*proof*⟩

lemma *inv_pair_neq*[*simp*, *dest*]:
False **if** *inv_pair (Pair_Storage k k x y)*
⟨*proof*⟩

lemma *inv_pair_P_D1*:
P m1 **if** *inv_pair (Pair_Storage k1 k2 m1 m2)*
⟨*proof*⟩

lemma *inv_pair_P_D2*:
P m2 **if** *inv_pair (Pair_Storage k1 k2 m1 m2)*
⟨*proof*⟩

lemma *inv_pair_domD*[*intro*]:
dom (map_of m1) ∩ dom (map_of m2) = {} **if** *inv_pair (Pair_Storage k1 k2 m1 m2)*
⟨*proof*⟩

lemma *mem_correct_pair*:
mem_correct lookup_pair update_pair inv_pair
⟨*proof*⟩

end

end

end

2.3 Indexing

theory *Indexing*
imports *Main*
begin

definition *injective* :: *nat* ⇒ (*'k* ⇒ *nat*) ⇒ *bool* **where**
injective size to_index ≡ ∀ *a b*.
to_index a = to_index b
∧ *to_index a < size*
∧ *to_index b < size*
⟶ *a = b*

for *size to_index*

lemma *index_mono*:
fixes *a b a0 b0* :: *nat*
assumes *a*: $a < a0$ **and** *b*: $b < b0$
shows $a * b0 + b < a0 * b0$
⟨*proof*⟩

lemma *index_eq_iff*:
fixes *a b c d b0* :: *nat*
assumes $b < b0$ $d < b0$ $a * b0 + b = c * b0 + d$
shows $a = c \wedge b = d$
⟨*proof*⟩

locale *prod_order_def* =
*order*0: *ord less_eq*0 *less*0 +
*order*1: *ord less_eq*1 *less*1
for *less_eq*0 *less*0 *less_eq*1 *less*1
begin

fun *less* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow$ *bool* **where**
less (*a*,*b*) (*c*,*d*) $\longleftrightarrow$ *less*0 *a c* $\wedge$ *less*1 *b d*

fun *less_eq* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow$ *bool* **where**
less_eq ab cd $\longleftrightarrow$ *less ab cd* $\vee$ *ab* = *cd*

end

locale *prod_order* =
*prod_order_def less_eq*0 *less*0 *less_eq*1 *less*1 +
*order*0: *order less_eq*0 *less*0 +
*order*1: *order less_eq*1 *less*1
for *less_eq*0 *less*0 *less_eq*1 *less*1
begin

sublocale *order less_eq less*
⟨*proof*⟩

end

locale *option_order* =
*order*0: *order less_eq*0 *less*0
for *less_eq*0 *less*0

begin

fun *less_eq_option* :: *'a option* ⇒ *'a option* ⇒ *bool* **where**
  less_eq_option None _ ⟷ *True*
| *less_eq_option* (*Some _*) *None* ⟷ *False*
| *less_eq_option* (*Some a*) (*Some b*) ⟷ *less_eq*0 *a b*

fun *less_option* :: *'a option* ⇒ *'a option* ⇒ *bool* **where**
  less_option ao bo ⟷ *less_eq_option ao bo* ∧ *ao* ≠ *bo*

sublocale *order less_eq_option less_option*
  ⟨*proof*⟩

end

datatype *'a bound* = *Bound* (*lower*: *'a*) (*upper*:*'a*)

definition *in_bound* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a bound* ⇒ *'a* ⇒ *bool* **where**
  in_bound less_eq less bound x ≡ *case bound of Bound l r* ⇒ *less_eq l x* ∧ *less x r* **for** *less_eq less*

locale *index_locale_def* = *ord less_eq less* **for** *less_eq less* :: *'a* ⇒ *'a* ⇒ *bool* +
  fixes *idx* :: *'a bound* ⇒ *'a* ⇒ *nat*
    and *size* :: *'a bound* ⇒ *nat*

locale *index_locale* = *index_locale_def* + *idx_ord*: *order* +
  assumes *idx_valid*: *in_bound less_eq less bound x* ⟹ *idx bound x* < *size bound*
    and *idx_inj* : ⟦*in_bound less_eq less bound x*; *in_bound less_eq less bound y*; *idx bound x* = *idx bound y*⟧ ⟹ *x* = *y*

locale *prod_index_def* =
  *index*0: *index_locale_def less_eq*0 *less*0 *idx*0 *size*0 +
  *index*1: *index_locale_def less_eq*1 *less*1 *idx*1 *size*1
  for *less_eq*0 *less*0 *idx*0 *size*0 *less_eq*1 *less*1 *idx*1 *size*1
begin

fun *idx* :: (*'a* × *'b*) *bound* ⇒ *'a* × *'b* ⇒ *nat* **where**
  idx (*Bound* (*l*0, *r*0) (*l*1, *r*1)) (*a*, *b*) = (*idx*0 (*Bound l*0 *l*1) *a*) ∗ (*size*1 (*Bound r*0 *r*1)) + *idx*1 (*Bound r*0 *r*1) *b*

fun *size* :: (*'a* × *'b*) *bound* ⇒ *nat* **where**

size (*Bound* (*l*0, *r*0) (*l*1, *r*1)) = *size*0 (*Bound l*0 *l*1) ∗ *size*1 (*Bound r*0 *r*1)

end

locale *prod_index* = *prod_index_def less_eq*0 *less*0 *idx*0 *size*0 *less_eq*1 *less*1 *idx*1 *size*1 +
*index*0: *index_locale less_eq*0 *less*0 *idx*0 *size*0 +
*index*1: *index_locale less_eq*1 *less*1 *idx*1 *size*1
for *less_eq*0 *less*0 *idx*0 *size*0 *less_eq*1 *less*1 *idx*1 *size*1
begin

sublocale *prod_order less_eq*0 *less*0 *less_eq*1 *less*1 ⟨*proof*⟩

sublocale *index_locale less_eq less idx size* ⟨*proof*⟩
end

locale *option_index* =
*index*0: *index_locale less_eq*0 *less*0 *idx*0 *size*0
for *less_eq*0 *less*0 *idx*0 *size*0
begin

fun *idx* :: *'a option bound* ⇒ *'a option* ⇒ *nat* **where**
idx (*Bound* (*Some l*) (*Some r*)) (*Some a*) = *idx*0 (*Bound l r*) *a*
| *idx* _ _ = *undefined*

end

locale *nat_index_def* = *ord* (≤) :: *nat* ⇒ *nat* ⇒ *bool* (<)
begin

fun *idx* :: *nat bound* ⇒ *nat* ⇒ *nat* **where**
idx (*Bound l* _) *i* = *i* − *l*

fun *size* :: *nat bound* ⇒ *nat* **where**
size (*Bound l r*) = *r* − *l*

sublocale *index_locale* (≤) (<) *idx size*
⟨*proof*⟩

end

locale *nat_index* = *nat_index_def* + *order* (≤) :: *nat* ⇒ *nat* ⇒ *bool* (<)

locale *int_index_def* = *ord* (≤) :: *int* ⇒ *int* ⇒ *bool* (<)
begin

fun *idx* :: *int bound* ⇒ *int* ⇒ *nat* **where**
  idx (*Bound l* _) *i* = *nat* (*i* − *l*)

fun *size* :: *int bound* ⇒ *nat* **where**
  size (*Bound l r*) = *nat* (*r* − *l*)

sublocale *index_locale* (≤) (<) *idx size*
⟨*proof*⟩

end

locale *int_index* = *int_index_def* + *order* (≤) :: *int* ⇒ *int* ⇒ *bool* (<)

class *index* =
  fixes *less_eq less* :: *'a* ⇒ *'a* ⇒ *bool*
    and *idx* :: *'a bound* ⇒ *'a* ⇒ *nat*
    and *size* :: *'a bound* ⇒ *nat*
  assumes *is_locale*: *index_locale less_eq less idx size*

locale *bounded_index* =
  fixes *bound* :: *'k* :: *index bound*
begin

interpretation *index_locale less_eq less idx size*
  ⟨*proof*⟩

definition *size* ≡ *index_class.size bound* **for** *size*

definition *checked_idx x* ≡ *if in_bound less_eq less bound x then idx bound x else size*

lemma *checked_idx_injective*:
  injective size checked_idx
  ⟨*proof*⟩
end

instantiation *nat* :: *index*
begin

interpretation *nat_index* ⟨*proof*⟩
thm *index_locale_axioms*

definition [*simp*]: *less_eq_nat* ≡ (≤) :: *nat* ⇒ *nat* ⇒ *bool*
definition [*simp*]: *less_nat* ≡ (<) :: *nat* ⇒ *nat* ⇒ *bool*
definition [*simp*]: *idx_nat* ≡ *idx*
definition *size_nat* **where** [*simp*]: *size_nat* ≡ *size*

instance ⟨*proof*⟩

end

instantiation *int* :: *index*
begin

interpretation *int_index* ⟨*proof*⟩
thm *index_locale_axioms*

definition [*simp*]: *less_eq_int* ≡ (≤) :: *int* ⇒ *int* ⇒ *bool*
definition [*simp*]: *less_int* ≡ (<) :: *int* ⇒ *int* ⇒ *bool*
definition [*simp*]: *idx_int* ≡ *idx*
definition [*simp*]: *size_int* ≡ *size*

lemmas *size_int* = *size.simps*

instance ⟨*proof*⟩
end

instantiation *prod* :: (*index*, *index*) *index*
begin

interpretation *prod_index*
  less_eq::'*a* ⇒ '*a* ⇒ *bool less idx size*
  less_eq::'*b* ⇒ '*b* ⇒ *bool less idx size*
  ⟨*proof*⟩
thm *index_locale_axioms*

definition [*simp*]: *less_eq_prod* ≡ *less_eq*
definition [*simp*]: *less_prod* ≡ *less*
definition [*simp*]: *idx_prod* ≡ *idx*
definition [*simp*]: *size_prod* ≡ *size* **for** *size_prod*

lemmas *size_prod* = *size.simps*

instance ⟨*proof*⟩

end

lemma *bound_int_simp*[*code*]:
bounded_index.size (*Bound* (*l*1, *l*2) (*u*1, *u*2)) = *nat* (*u*1 − *l*1) ∗ *nat* (*u*2 − *l*2)
⟨*proof*⟩

lemmas [*code*] = *bounded_index.size_def bounded_index.checked_idx_def*

lemmas [*code*] =
nat_index_def.size.simps
nat_index_def.idx.simps

lemmas [*code*] =
int_index_def.size.simps
int_index_def.idx.simps

lemmas [*code*] =
prod_index_def.size.simps
prod_index_def.idx.simps

lemmas [*code*] =
prod_order_def.less_eq.simps
prod_order_def.less.simps

lemmas *index_size_defs* =
prod_index_def.size.simps int_index_def.size.simps nat_index_def.size.simps bounded_index.size_def

end

2.4 Heap Memory Implementations

theory *Memory_Heap*
imports *State_Heap DP_CRelVH Pair_Memory HOL−Eisbach.Eisbach ../Indexing*
begin

Move

abbreviation *result_of c h* ≡ *fst* (*the* (*execute c h*))
abbreviation *heap_of c h* ≡ *snd* (*the* (*execute c h*))

lemma *map_emptyI*:
m ⊆*m* *Map.empty* **if** ⋀ *x*. *m x* = *None*

⟨*proof*⟩

lemma *result_of_return*[*simp*]:
result_of (*Heap_Monad.return x*) *h* = *x*
⟨*proof*⟩

lemma *get_result_of_lookup*:
result_of (!*r*) *heap* = *x* **if** *Ref.get heap r* = *x*
⟨*proof*⟩

context
fixes *size* :: *nat*
and *to_index* :: ('*k*2 :: *heap*) ⇒ *nat*
begin

definition
mem_empty = (*Array.new size* (*None* :: ('*v* :: *heap*) *option*))

lemma *success_empty*[*intro*]:
success mem_empty heap
⟨*proof*⟩

lemma *length_mem_empty*:
Array.length
(*heap_of* (*mem_empty*:: ((*'b* :: *heap*) *option array*) *Heap*) *h*)
(*result_of* (*mem_empty* :: (*'b option array*) *Heap*) *h*) = *size*
⟨*proof*⟩

lemma *nth_mem_empty*:
result_of
(*Array.nth* (*result_of* (*mem_empty* :: (*'b option array*) *Heap*) *h*) *i*)
(*heap_of* (*mem_empty* :: ((*'b* :: *heap*) *option array*) *Heap*) *h*) = *None*
if *i* < *size*
⟨*proof*⟩

context
fixes *mem* :: (*'v* :: *heap*) *option array*
begin

definition
mem_lookup k = (*let i* = *to_index k in*
if i < *size then Array.nth mem i else return None*
)

definition
mem_update k v = (*let i* = *to_index k in*
if i < *size then* (*Array.upd i* (*Some v*) *mem* ⤜ (λ _. *return* ()))
else return ()
)

context assumes *injective*: *injective size to_index*
begin

interpretation *heap_correct* λ*heap. Array.length heap mem* = *size mem_update mem_lookup*
⟨*proof*⟩

lemmas *mem_heap_correct* = *heap_correct_axioms*

context
assumes [*simp*]: *mem* = *result_of mem_empty Heap.empty*
begin

interpretation *heap_correct_empty*
λ*heap. Array.length heap mem* = *size mem_update mem_lookup*
heap_of (*mem_empty* :: *'v option array Heap*) *Heap.empty*
⟨*proof*⟩

lemmas *array_heap_emptyI* = *heap_correct_empty_axioms*

context
fixes *dp* :: *'k2* ⇒ *'v*
begin

interpretation *dp_consistency_heap_empty*
λ*heap. Array.length heap mem* = *size mem_update mem_lookup dp*
heap_of (*mem_empty* :: *'v option array Heap*) *Heap.empty*
⟨*proof*⟩

lemmas *array_consistentI* = *dp_consistency_heap_empty_axioms*

end

end

end

end

lemma *execute_bind_success'*:
assumes *success f h execute* $(f \gg\!= g)$ h = *Some* (y, h'')
obtains x h' **where** *execute f h* = *Some* (x, h') *execute* $(g\ x)$ h' = *Some* (y, h'')
⟨*proof*⟩

lemma *success_bind_I*:
assumes *success f h*
and $\bigwedge x\ h'$. *execute f h* = *Some* $(x, h') \Longrightarrow$ *success* $(g\ x)$ h'
shows *success* $(f \gg\!= g)$ h
⟨*proof*⟩

definition
alloc_pair a b ≡ *do* {
*r*1 ← *ref a*;
*r*2 ← *ref b*;
return (*r*1, *r*2)
}

lemma *alloc_pair_alloc*:
Ref.get heap' *r*1 = *a* *Ref.get heap'* *r*2 = *b*
if *execute* (*alloc_pair a b*) *heap* = *Some* ((*r*1, *r*2), *heap'*)
⟨*proof*⟩

lemma *alloc_pairD*1:
r =!= *r*1 ∧ r =!= *r*2 ∧ *Ref.present heap'* r
if *execute* (*alloc_pair a b*) *heap* = *Some* ((*r*1, *r*2), *heap'*) *Ref.present heap* r
⟨*proof*⟩

lemma *alloc_pairD*2:
*r*1 =!= *r*2 ∧ *Ref.present heap'* *r*2 ∧ *Ref.present heap'* *r*1
if *execute* (*alloc_pair a b*) *heap* = *Some* ((*r*1, *r*2), *heap'*)
⟨*proof*⟩

lemma *alloc_pairD*3:
Array.present heap' r
if *execute* (*alloc_pair a b*) *heap* = *Some* ((*r*1, *r*2), *heap'*) *Array.present heap* r
⟨*proof*⟩

lemma *alloc_pairD*4:

Ref.get heap′ r = x
if *execute (alloc_pair a b) heap = Some ((r1, r2), heap′)*
Ref.get heap r = x Ref.present heap r
⟨*proof*⟩

lemma *alloc_pair_array_get*:
Array.get heap′ r = x
if *execute (alloc_pair a b) heap = Some ((r1, r2), heap′) Array.get heap r = x*
⟨*proof*⟩

lemma *alloc_pair_array_length*:
Array.length heap′ r = Array.length heap r
if *execute (alloc_pair a b) heap = Some ((r1, r2), heap′)*
⟨*proof*⟩

lemma *alloc_pair_nth*:
result_of (Array.nth r i) heap′ = result_of (Array.nth r i) heap
if *execute (alloc_pair a b) heap = Some ((r1, r2), heap′)*
⟨*proof*⟩

lemma *succes_alloc_pair*[*intro*]:
success (alloc_pair a b) heap
⟨*proof*⟩

definition
init_state_inner k1 k2 m1 m2 ≡ do {
(k_ref1, k_ref2) ← alloc_pair k1 k2;
(m_ref1, m_ref2) ← alloc_pair m1 m2;
return (k_ref1, k_ref2, m_ref1, m_ref2)
}

lemma *init_state_inner_alloc*:
assumes
execute (init_state_inner k1 k2 m1 m2) heap = Some ((k_ref1, k_ref2, m_ref1, m_ref2), heap′)
shows
Ref.get heap′ k_ref1 = k1 Ref.get heap′ k_ref2 = k2
Ref.get heap′ m_ref1 = m1 Ref.get heap′ m_ref2 = m2
⟨*proof*⟩

lemma *init_state_inner_distinct*:
assumes

execute (*init_state_inner k1 k2 m1 m2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
shows
m_ref1 =!= *m_ref2* ∧ *m_ref1* =!= *k_ref1* ∧ *m_ref1* =!= *k_ref2* ∧ *m_ref2* =!= *k_ref1*
∧ *m_ref2* =!= *k_ref2* ∧ *k_ref1* =!= *k_ref2*
⟨*proof*⟩

lemma *init_state_inner_present*:
assumes
execute (*init_state_inner k1 k2 m1 m2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
shows
Ref.present heap' k_ref1 Ref.present heap' k_ref2
Ref.present heap' m_ref1 Ref.present heap' m_ref2
⟨*proof*⟩

lemma *inite_state_inner_present'*:
assumes
execute (*init_state_inner k1 k2 m1 m2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
Array.present heap a
shows
Array.present heap' a
⟨*proof*⟩

lemma *succes_init_state_inner*[*intro*]:
success (*init_state_inner k1 k2 m1 m2*) *heap*
⟨*proof*⟩

lemma *init_state_inner_nth*:
result_of (*Array.nth r i*) *heap'* = *result_of* (*Array.nth r i*) *heap*
if *execute* (*init_state_inner k1 k2 m1 m2*) *heap* = *Some* ((*r1*, *r2*), *heap'*)
⟨*proof*⟩

definition
init_state k1 k2 ≡ *do* {
m1 ← *mem_empty*;
m2 ← *mem_empty*;
init_state_inner k1 k2 m1 m2
}

lemma *succes_init_state*[*intro*]:
success (*init_state k1 k2*) *heap*

⟨*proof*⟩

definition
inv_distinct k_ref1 k_ref2 m_ref1 m_ref2 ≡
m_ref1 =!= *m_ref2* ∧ *m_ref1* =!= *k_ref1* ∧ *m_ref1* =!= *k_ref2* ∧ *m_ref2* =!= *k_ref1*
∧ *m_ref2* =!= *k_ref2* ∧ *k_ref1* =!= *k_ref2*

lemma *init_state_distinct*:
assumes
execute (*init_state k1 k2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
shows
inv_distinct k_ref1 k_ref2 m_ref1 m_ref2
⟨*proof*⟩

lemma *init_state_present*:
assumes
execute (*init_state k1 k2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
shows
Ref.present heap' k_ref1 Ref.present heap' k_ref2
Ref.present heap' m_ref1 Ref.present heap' m_ref2
⟨*proof*⟩

lemma *empty_present*:
Array.present h' x **if** *execute mem_empty heap* = *Some* (*x*, *h'*)
⟨*proof*⟩

lemma *empty_present'*:
Array.present h' a **if** *execute mem_empty heap* = *Some* (*x*, *h'*) *Array.present heap a*
⟨*proof*⟩

lemma *init_state_present2*:
assumes
execute (*init_state k1 k2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
shows
Array.present heap' (*Ref.get heap' m_ref1*) *Array.present heap'* (*Ref.get heap' m_ref2*)
⟨*proof*⟩

lemma *init_state_neq*:
assumes
execute (*init_state k1 k2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
shows
Ref.get heap' m_ref1 =!!= *Ref.get heap' m_ref2*
⟨*proof*⟩

lemma *present_alloc_get*:
Array.get heap' a = *Array.get heap a*
if *Array.alloc xs heap* = (*a'*, *heap'*) *Array.present heap a*
⟨*proof*⟩

lemma *init_state_length*:
assumes
execute (*init_state k1 k2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap'*)
shows
Array.length heap' (*Ref.get heap' m_ref1*) = *size*
Array.length heap' (*Ref.get heap' m_ref2*) = *size*
⟨*proof*⟩

context
fixes *key1* :: *'k* ⇒ (*'k1* :: *heap*) **and** *key2* :: *'k* ⇒ *'k2*
and *m_ref1 m_ref2* :: (*'v* :: *heap*) *option array ref*
and *k_ref1 k_ref2* :: (*'k1* :: *heap*) *ref*
begin

We assume that look-ups happen on the older row, so this is biased towards the second entry.

definition
lookup_pair k = *do* {
let k' = *key1 k*;
k2 ← !*k_ref2*;
if k' = *k2 then*
do {
m2 ← !*m_ref2*;
mem_lookup m2 (*key2 k*)
}
else
do {
k1 ← !*k_ref1*;
if k' = *k1 then*
do {

```
      m1 ← !m_ref1;
      mem_lookup m1 (key2 k)
    }
  else
    return None
 }
}
```

We assume that updates happen on the newer row, so this is biased towards the first entry.

definition

```
update_pair k v = do {
 let k' = key1 k;
  k1 ← !k_ref1;
  if k' = k1 then do {
    m ← !m_ref1;
    mem_update m (key2 k) v
  }
  else do {
    k2 ← !k_ref2;
    if k' = k2 then do {
      m ← !m_ref2;
      mem_update m (key2 k) v
    }
    else do {
      do {
        k1 ← !k_ref1;
        m ← mem_empty;
        m1 ← !m_ref1;
        k_ref2 := k1;
        k_ref1 := k';
        m_ref2 := m1;
        m_ref1 := m
      }
    ;
    m ← !m_ref1;
    mem_update m (key2 k) v
  }
 }
}
```

definition

```
inv_pair_weak heap = (
  let
    m1 = Ref.get heap m_ref1;
    m2 = Ref.get heap m_ref2
  in Array.length heap m1 = size ∧ Array.length heap m2 = size
    ∧ Ref.present heap k_ref1 ∧ Ref.present heap k_ref2
    ∧ Ref.present heap m_ref1 ∧ Ref.present heap m_ref2
    ∧ Array.present heap m1 ∧ Array.present heap m2
    ∧ m1 =!!= m2
)
```

definition
inv_pair heap ≡ *inv_pair_weak heap* ∧ *inv_distinct k_ref1 k_ref2 m_ref1 m_ref2*

lemma *init_state_inv*:
assumes
execute (*init_state k1 k2*) *heap* = *Some* ((*k_ref1*, *k_ref2*, *m_ref1*, *m_ref2*), *heap′*)
shows *inv_pair_weak heap′*
⟨*proof*⟩

lemma *inv_pair_lengthD1*:
Array.length heap (*Ref.get heap m_ref1*) = *size* **if** *inv_pair_weak heap*
⟨*proof*⟩

lemma *inv_pair_lengthD2*:
Array.length heap (*Ref.get heap m_ref2*) = *size* **if** *inv_pair_weak heap*
⟨*proof*⟩

lemma *inv_pair_presentD*:
Array.present heap (*Ref.get heap m_ref1*) *Array.present heap* (*Ref.get heap m_ref2*)
if *inv_pair_weak heap*
⟨*proof*⟩

lemma *inv_pair_presentD2*:
Ref.present heap m_ref1 Ref.present heap m_ref2
Ref.present heap k_ref1 Ref.present heap k_ref2
if *inv_pair_weak heap*
⟨*proof*⟩

lemma *inv_pair_not_eqD*:

Ref.get heap m_ref1 =!!= *Ref.get heap m_ref2* **if** *inv_pair_weak heap*
⟨*proof*⟩

definition *lookup1 k* ≡ *state_of* (*do* {*m* ← !*m_ref1*; *mem_lookup m k*})

definition *lookup2 k* ≡ *state_of* (*do* {*m* ← !*m_ref2*; *mem_lookup m k*})

definition *update1 k v* ≡ *state_of* (*do* {*m* ← !*m_ref1*; *mem_update m k v*})

definition *update2 k v* ≡ *state_of* (*do* {*m* ← !*m_ref2*; *mem_update m k v*})

definition *move12 k* ≡ *state_of* (*do* {
k1 ← !*k_ref1*;
m ← *mem_empty*;
m1 ← !*m_ref1*;
k_ref2 := *k1*;
k_ref1 := *k*;
m_ref2 := *m1*;
m_ref1 := *m*
})

definition *get_k1* ≡ *state_of* (!*k_ref1*)

definition *get_k2* ≡ *state_of* (!*k_ref2*)

lemma *run_state_state_of*[*simp*]:
State_Monad.run_state (*state_of p*) *m* = *the* (*execute p m*)
⟨*proof*⟩

context assumes *injective*: *injective size to_index*
begin

context
assumes *inv_distinct*: *inv_distinct k_ref1 k_ref2 m_ref1 m_ref2*
begin

lemma *disjoint*[*simp*]:
m_ref1 =!= *m_ref2* *m_ref1* =!= *k_ref1* *m_ref1* =!= *k_ref2*
m_ref2 =!= *k_ref1* *m_ref2* =!= *k_ref2*
k_ref1 =!= *k_ref2*
⟨*proof*⟩

lemmas [*simp*] = *disjoint*[*THEN noteq_sym*]

lemma [*simp*]:
Array.get (*snd* (*Array.alloc xs heap*)) *a* = *Array.get heap a* **if** *Array.present heap a*
⟨*proof*⟩

lemma [*simp*]:
Ref.get (*snd* (*Array.alloc xs heap*)) *r* = *Ref.get heap r* **if** *Ref.present heap r*
⟨*proof*⟩

lemma *alloc_present*:
Array.present (*snd* (*Array.alloc xs heap*)) *a* **if** *Array.present heap a*
⟨*proof*⟩

lemma *alloc_present'*:
Ref.present (*snd* (*Array.alloc xs heap*)) *r* **if** *Ref.present heap r*
⟨*proof*⟩

lemma *length_get_upd*[*simp*]:
length (*Array.get* (*Array.update a i x heap*) *r*) = *length* (*Array.get heap r*)
⟨*proof*⟩

method *solve*1 =
(*frule inv_pair_lengthD*1, *frule inv_pair_lengthD*2, *frule inv_pair_not_eqD*) *?*,
auto split: *if_split_asm dest*: *Array.noteq_sym*

interpretation *pair*: *pair_mem lookup*1 *lookup*2 *update*1 *update*2 *move*12 *get_k*1 *get_k*2 *inv_pair_weak*
⟨*proof*⟩

lemmas *mem_correct_pair* = *pair.mem_correct_pair*

definition
*mem_lookup*1 *k* = *do* {*m* ← !*m_ref*1; *mem_lookup m k*}

definition
*mem_lookup*2 *k* = *do* {*m* ← !*m_ref*2; *mem_lookup m k*}

definition *get_k*1′ ≡ !*k_ref*1

definition *get_k*2′ ≡ !*k_ref*2

definition *update*1′ *k v* ≡ *do* {*m* ← !*m_ref*1; *mem_update m k v*}

definition *update*2′ *k v* ≡ *do* {*m* ← !*m_ref*2; *mem_update m k v*}

definition *move*12′ *k* ≡ *do* {
  *k*1 ← !*k_ref*1;
  m ← *mem_empty*;
  *m*1 ← !*m_ref*1;
  *k_ref*2 := *k*1;
  *k_ref*1 := *k*;
  *m_ref*2 := *m*1;
  *m_ref*1 := *m*
 }

interpretation *heap_mem_defs inv_pair_weak lookup_pair update_pair*
⟨*proof*⟩

lemma *rel_state_ofI*:
 rel_state (=) (*state_of m*) *m* **if**
 ∀ *heap. inv_pair_weak heap* ⟶ *success m heap*
 lift_p inv_pair_weak m
 ⟨*proof*⟩

lemma *inv_pair_iff*:
 inv_pair_weak = *inv_pair*
 ⟨*proof*⟩

lemma *lift_p_inv_pairI*:
 State_Heap.lift_p inv_pair m **if** *State_Heap.lift_p inv_pair_weak m*
 ⟨*proof*⟩

lemma *lift_p_success*:
 State_Heap.lift_p inv_pair_weak m
 if *DP_CRelVS.lift_p inv_pair_weak* (*state_of m*) ∀ *heap. inv_pair_weak heap* ⟶ *success m heap*
 ⟨*proof*⟩

lemma *rel_state_ofI*2:
 rel_state (=) (*state_of m*) *m* **if**
 ∀ *heap. inv_pair_weak heap* ⟶ *success m heap*
 DP_CRelVS.lift_p inv_pair_weak (*state_of m*)
 ⟨*proof*⟩

context
includes *lifting_syntax*
begin

lemma [*transfer_rule*]:
((=) ===> *rel_state* (=)) *move*12 *move*12′
⟨*proof*⟩

lemma [*transfer_rule*]:
((=) ===> *rel_state* (*rel_option* (=))) *lookup*1 *mem_lookup*1
⟨*proof*⟩

lemma [*transfer_rule*]:
((=) ===> *rel_state* (*rel_option* (=))) *lookup*2 *mem_lookup*2
⟨*proof*⟩

lemma [*transfer_rule*]:
rel_state (=) *get_k*1 *get_k*1′
⟨*proof*⟩

lemma [*transfer_rule*]:
rel_state (=) *get_k*2 *get_k*2′
⟨*proof*⟩

lemma [*transfer_rule*]:
((=) ===> (=) ===> *rel_state* (=)) *update*1 *update*1′
⟨*proof*⟩

lemma [*transfer_rule*]:
((=) ===> (=) ===> *rel_state* (=)) *update*2 *update*2′
⟨*proof*⟩

lemma [*transfer_rule*]:
((=) ===> *rel_state* (*rel_option* (=))) *lookup*1 *mem_lookup*1
⟨*proof*⟩

lemma *rel_state_lookup*:
((=) ===> *rel_state* (=)) *pair.lookup_pair lookup_pair*
⟨*proof*⟩

lemma *rel_state_update*:
((=) ===> (=) ===> *rel_state* (=)) *pair.update_pair update_pair*
⟨*proof*⟩

interpretation *mem*: *heap_mem_defs pair.inv_pair lookup_pair update_pair*
⟨*proof*⟩

lemma *inv_pairD*:
inv_pair_weak heap **if** *pair.inv_pair heap*
⟨*proof*⟩

lemma *mem_rel_state_ofI*:
mem.rel_state (=) *m′ m* **if**
rel_state (=) *m′ m*
⋀ *heap*. *pair.inv_pair heap* ⟹
(*case State_Monad.run_state m′ heap of* (_, *heap*) ⇒ *inv_pair_weak heap* ⟶ *pair.inv_pair heap*)
⟨*proof*⟩

lemma *mem_rel_state_ofI′*:
mem.rel_state (=) *m′ m* **if**
rel_state (=) *m′ m*
DP_CRelVS.lift_p pair.inv_pair m′
⟨*proof*⟩

context
assumes *keys*: ∀ *k k′*. *key1 k* = *key1 k′* ∧ *key2 k* = *key2 k′* ⟶ *k* = *k′*
begin

interpretation *mem_correct pair.lookup_pair pair.update_pair pair.inv_pair*
⟨*proof*⟩

lemma *rel_state_lookup′*:
((=) ===> *mem.rel_state* (=)) *pair.lookup_pair lookup_pair*
⟨*proof*⟩

lemma *rel_state_update′*:
((=) ===> (=) ===> *mem.rel_state* (=)) *pair.update_pair update_pair*
⟨*proof*⟩

interpretation *heap_correct pair.inv_pair update_pair lookup_pair*
⟨*proof*⟩

lemmas *heap_correct_pairI* = *heap_correct_axioms*

lemma *mem_rel_state_resultD*:
result_of m heap = *fst* (*run_state m′ heap*) **if** *mem.rel_state* (=) *m′ m*

pair.inv_pair heap
⟨*proof*⟩

lemma *map_of_heap_eq*:
mem.map_of_heap heap = *pair.pair.map_of heap* **if** *pair.inv_pair heap*
⟨*proof*⟩

context
fixes *k*1 *k*2 *heap heap*′
assumes *init*: *execute* (*init_state k*1 *k*2) *heap* = *Some* ((*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2), *heap*′)
begin

lemma *init_state_empty*1:
*pair.mem*1.*map_of heap*′ *k* = *None*
⟨*proof*⟩

lemma *init_state_empty*2:
*pair.mem*2.*map_of heap*′ *k* = *None*
⟨*proof*⟩

lemma
shows *init_state_k*1: *result_of* (!*k_ref*1) *heap*′ = *k*1
and *init_state_k*2: *result_of* (!*k_ref*2) *heap*′ = *k*2
⟨*proof*⟩

context
assumes *neq*: *k*1 ≠ *k*2
begin

lemma *init_state_inv*′:
pair.inv_pair heap′
⟨*proof*⟩

lemma *init_state_empty*:
pair.pair.map_of heap′ ⊆$_m$ *Map.empty*
⟨*proof*⟩

interpretation *heap_correct_empty pair.inv_pair update_pair lookup_pair heap*′
⟨*proof*⟩

lemmas *heap_correct_empty_pairI* = *heap_correct_empty_axioms*

context
fixes *dp* :: $'k \Rightarrow 'v$
begin

interpretation *dp_consistency_heap_empty*
pair.inv_pair update_pair lookup_pair dp heap'
⟨*proof*⟩

lemmas *consistent_empty_pairI* = *dp_consistency_heap_empty_axioms*

end

end

end

end

end

end

end

end

end

end

2.5 Tool Setup

theory *Transform_Cmd*
imports
../Pure_Monad
../state_monad/DP_CRelVS
../heap_monad/DP_CRelVH
keywords
memoize_fun :: *thy_decl*
and *monadifies* :: *thy_decl*
and *memoize_correct* :: *thy_goal*
and *with_memory* :: *quasi_command*
and *default_proof* :: *quasi_command*
begin

⟨*ML*⟩

end

2.6 Bottom-Up Computation

theory *Bottom_Up_Computation*
imports *../state_monad/Memory ../state_monad/DP_CRelVS*
begin

fun *iterate_state* **where**
iterate_state f [] = *State_Monad.return* () |
iterate_state f (*x* # *xs*) = *do* {*f x*; *iterate_state f xs*}

locale *iterator_defs* =
fixes *cnt* :: *'a* ⇒ *bool* **and** *nxt* :: *'a* ⇒ *'a*
begin

definition
iter_state f ≡
wfrec
{(*nxt x*, *x*) | *x*. *cnt x*}
(λ *rec x*. *if cnt x then do* {*f x*; *rec* (*nxt x*)} *else State_Monad.return*
())

definition
iterator_to_list ≡
wfrec {(*nxt x*, *x*) | *x*. *cnt x*} (λ *rec x*. *if cnt x then x* # *rec* (*nxt x*) *else*
[])

end

locale *iterator* = *iterator_defs* +
fixes *sizef* :: *'a* ⇒ *nat*
assumes *terminating*:
finite {*x*. *cnt x*} ∀ *x*. *cnt x* ⟶ *sizef x* < *sizef* (*nxt x*)
begin

lemma *admissible*:
adm_wf
{(*nxt x*, *x*) | *x*. *cnt x*}
(λ *rec x*. *if cnt x then do* {*f x*; *rec* (*nxt x*)} *else State_Monad.return*

())
⟨*proof*⟩

lemma *wellfounded*:
wf {(*nxt x*, *x*) | *x*. *cnt x*} (**is** *wf ?S*)
⟨*proof*⟩

lemma *iter_state_unfold*:
iter_state f x = (*if cnt x then do* {*f x*; *iter_state f* (*nxt x*)} *else State_Monad.return*
())
⟨*proof*⟩

lemma *iterator_to_list_unfold*:
iterator_to_list x = (*if cnt x then x* # *iterator_to_list* (*nxt x*) *else* [])
⟨*proof*⟩

lemma *iter_state_iterate_state*:
iter_state f x = *iterate_state f* (*iterator_to_list x*)
⟨*proof*⟩

end

context *dp_consistency*
begin

context
includes *lifting_syntax*
begin

lemma *crel_vs_iterate_state*:
crel_vs (=) () (*iterate_state f xs*) **if** ((=) ===>$_T$ *R*) *g f*
⟨*proof*⟩

lemma *crel_vs_bind_ignore*:
crel_vs R a (*do* {*d*; *b*}) **if** *crel_vs R a b crel_vs S c d*
⟨*proof*⟩

lemma *crel_vs_iterate_and_compute*:
assumes ((=) ===>$_T$ *R*) *g f*
shows *crel_vs R* (*g x*) (*do* {*iterate_state f xs*; *f x*})
⟨*proof*⟩

end

end

locale *dp_consistency_iterator* =
  dp_consistency lookup update + *iterator cnt nxt sizef*
  for *lookup* :: *'a* ⇒ (*'b*, *'c option*) *state* **and** *update*
    and *cnt* :: *'a* ⇒ *bool* **and** *nxt* **and** *sizef*
begin

lemma *crel_vs_iter_and_compute*:
  assumes ((=) ===>$_T$ *R*) *g f*
  shows *crel_vs R* (*g x*) (*do* {*iter_state f y*; *f x*})
  ⟨*proof*⟩

lemma *consistentDP_iter_and_compute*:
  assumes *consistentDP f*
  shows *crel_vs* (=) (*dp x*) (*do* {*iter_state f y*; *f x*})
  ⟨*proof*⟩

end

locale *dp_consistency_iterator_empty* =
  dp_consistency_iterator + *dp_consistency_empty*
begin

lemma *memoized*:
  dp x = *fst* (*run_state* (*do* {*iter_state f y*; *f x*}) *empty*) **if** *consistentDP f*
  ⟨*proof*⟩

lemma *cmem_result*:
  cmem (*snd* (*run_state* (*do* {*iter_state f y*; *f x*}) *empty*)) **if** *consistentDP*
f
  ⟨*proof*⟩

end

lemma *dp_consistency_iterator_emptyI*:
  dp_consistency_iterator_empty P lookup update cnt
    nxt sizef empty
  if *dp_consistency_empty lookup update P empty*
     iterator cnt nxt sizef
   for *empty*
  ⟨*proof*⟩

context

fixes $m :: nat$ — Width of a row
and $n :: nat$ — Number of rows
begin

lemma *table_iterator_up*:
iterator
$(\lambda\ (x,\ y).\ x \leq n \wedge y \leq m)$
$(\lambda\ (x,\ y).\ \mathit{if}\ y < m\ \mathit{then}\ (x,\ y + 1)\ \mathit{else}\ (x + 1,\ 0))$
$(\lambda\ (x,\ y).\ x * (m + 1) + y)$
⟨*proof*⟩

lemma *table_iterator_down*:
iterator
$(\lambda\ (x,\ y).\ x \leq n \wedge y \leq m \wedge x > 0)$
$(\lambda\ (x,\ y).\ \mathit{if}\ y > 0\ \mathit{then}\ (x,\ y - 1)\ \mathit{else}\ (x - 1,\ m))$
$(\lambda\ (x,\ y).\ (n - x) * (m + 1) + (m - y))$
⟨*proof*⟩

end

end
theory *Bottom_Up_Computation_Heap*
imports *../state_monad/Bottom_Up_Computation ../heap_monad/DP_CRelVH*
begin

definition (**in** *iterator_defs*)
iter_heap f ≡
wfrec
$\{(nxt\ x,\ x) \mid x.\ cnt\ x\}$
$(\lambda\ rec\ x.\ \mathit{if}\ cnt\ x\ \mathit{then}\ do\ \{f\ x;\ rec\ (nxt\ x)\}\ \mathit{else}\ return\ ())$

lemma (**in** *iterator*) *iter_heap_unfold*:
iter_heap f x = (*if cnt x then do* $\{f\ x;$ *iter_heap f* (*nxt x*)$\}$ *else return* ())
⟨*proof*⟩

locale *dp_consistency_iterator_heap* =
dp_consistency_heap P update lookup dp + *iterator cnt nxt sizef*
for *lookup* :: $'a \Rightarrow ('c\ option)\ Heap$ **and** *update* **and** *P dp*
and *cnt* :: $'a \Rightarrow bool$ **and** *nxt* **and** *sizef*
begin

context
includes *lifting_syntax*
begin

term *iter_heap*

term *crel_vs*

lemma *crel_vs_iterate_state*:
crel_vs (=) () (*iter_heap f x*) **if** ((=) ===> *crel_vs R*) *g f*
⟨*proof*⟩

lemma *crel_vs_bind_ignore*:
crel_vs R a (*do* {*d*; *b*}) **if** *crel_vs R a b crel_vs S c d*
⟨*proof*⟩

lemma *crel_vs_iter_and_compute*:
assumes ((=) ===> *crel_vs R*) *g f*
shows *crel_vs R* (*g x*) (*do* {*iter_heap f y*; *f x*})
⟨*proof*⟩

lemma *consistent_DP_iter_and_compute*:
assumes *consistentDP f*
shows *consistentDP* (λ *x*. *do* {*iter_heap f y*; *f x*})
⟨*proof*⟩

end

end

end

2.7 Setup for the Heap Monad

theory *Solve_Cong*
imports *Main HOL−Eisbach.Eisbach*
begin

Method for solving trivial equalities with congruence reasoning

named_theorems *cong_rules*

method *solve_cong* **methods** *solve* =
rule HOL.refl |
rule cong_rules; *solve_cong solve* |
solve; *fail*

end

theory *Heap_Main*
imports
../heap_monad/Memory_Heap
../transform/Transform_Cmd
Bottom_Up_Computation_Heap
../util/Solve_Cong
begin

context includes *heap_monad_syntax* **begin**

thm *if_cong*
lemma *ifT_cong*:
assumes $b = c$ $c \Longrightarrow x = u$ $\neg c \Longrightarrow y = v$
shows *Heap_Monad_Ext.if*$_T$ $\langle b \rangle$ x y = *Heap_Monad_Ext.if*$_T$ $\langle c \rangle$ u v
⟨*proof*⟩

lemma *return_app_return_cong*:
assumes $f\ x = g\ y$
shows $\langle f \rangle$. $\langle x \rangle = \langle g \rangle$. $\langle y \rangle$
⟨*proof*⟩

lemmas [*fundef_cong*] =
return_app_return_cong
ifT_cong
end
memoize_fun *comp*$_T$: *comp* **monadifies** (*heap*) *comp_def*
thm *comp*$_T$*'.simps*
lemma (**in** *dp_consistency_heap*) **shows** *comp*$_T$*_transfer*[*transfer_rule*]:
crel_vs ((*R*1 ===>$_T$ *R*2) ===>$_T$ (*R*0 ===>$_T$ *R*1) ===>$_T$ (*R*0 ===>$_T$ *R*2)) *comp* *comp*$_T$
⟨*proof*⟩

memoize_fun *map*$_T$: *map* **monadifies** (*heap*) *list.map*
lemma (**in** *dp_consistency_heap*) *map*$_T$*_transfer*[*transfer_rule*]:
crel_vs ((*R*0 ===>$_T$ *R*1) ===>$_T$ *list_all*2 *R*0 ===>$_T$ *list_all*2 *R*1) *map* *map*$_T$
⟨*proof*⟩

memoize_fun *fold*$_T$: *fold* **monadifies** (*heap*) *fold.simps*
lemma (**in** *dp_consistency_heap*) *fold*$_T$*_transfer*[*transfer_rule*]:
crel_vs ((*R*0 ===>$_T$ *R*1 ===>$_T$ *R*1) ===>$_T$ *list_all*2 *R*0 ===>$_T$ *R*1 ===>$_T$ *R*1) *fold* *fold*$_T$
⟨*proof*⟩

context includes *heap_monad_syntax* **begin**

thm *map_cong*
lemma *mapT_cong*:
assumes $xs = ys$ $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
shows $map_T\ .\ \langle f \rangle\ .\ \langle xs \rangle = map_T\ .\ \langle g \rangle\ .\ \langle ys \rangle$
⟨*proof*⟩

thm *fold_cong*
lemma *foldT_cong*:
assumes $xs = ys$ $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
shows $fold_T\ .\ \langle f \rangle\ .\ \langle xs \rangle = fold_T\ .\ \langle g \rangle\ .\ \langle ys \rangle$
⟨*proof*⟩

lemma *abs_unit_cong*:

assumes $x = y$
shows $(\lambda\_::unit.\ x) = (\lambda\_.\ y)$
⟨*proof*⟩

lemma *arg_cong4*:
$f\ a\ b\ c\ d = f\ a'\ b'\ c'\ d'$ **if** $a = a'\ b = b'\ c = c'\ d = d'$
⟨*proof*⟩

lemmas [*fundef_cong*, *cong_rules*] =
return_app_return_cong
ifT_cong
mapT_cong
foldT_cong
abs_unit_cong
lemmas [*cong_rules*] =
arg_cong4[**where** f = *heap_mem_defs.checkmem*]
arg_cong2[**where** f = *fun_app_lifted*]
end

context *dp_consistency_heap* **begin**
context includes *lifting_syntax heap_monad_syntax* **begin**

named_theorems *dp_match_rule*

thm *if_cong*
lemma *if*$_T$*_cong2*:

assumes *Rel* (=) *b c c* ⟹ *Rel* (*crel_vs R*) *x* x_T ¬*c* ⟹ *Rel* (*crel_vs R*) *y* y_T
shows *Rel* (*crel_vs R*) (*if* (*Wrap b*) *then x else y*) (*Heap_Monad_Ext.if*$_T$ ⟨*c*⟩ x_T y_T)
⟨*proof*⟩

lemma *map*$_T$*_cong*2:
assumes
is_equality R
Rel R xs ys
⋀*x*. *x*∈*set ys* ⟹ *Rel* (*crel_vs S*) (*f x*) (f_T′ *x*)
shows *Rel* (*crel_vs* (*list_all*2 *S*)) (*App* (*App map* (*Wrap f*)) (*Wrap xs*)) (*map*$_T$. ⟨f_T′⟩ . ⟨*ys*⟩)
⟨*proof*⟩

lemma *fold*$_T$*_cong*2:
assumes
is_equality R
Rel R xs ys
⋀*x*. *x*∈*set ys* ⟹ *Rel* (*crel_vs* (*S* ===> *crel_vs S*)) (*f x*) (f_T′ *x*)
shows
Rel (*crel_vs* (*S* ===> *crel_vs S*)) (*fold f xs*) (*fold*$_T$. ⟨f_T′⟩ . ⟨*ys*⟩)
⟨*proof*⟩

lemma *refl*2:
is_equality R ⟹ *Rel R x x*
⟨*proof*⟩

lemma *rel_fun*2:
assumes *is_equality R*0 ⋀*x*. *Rel R*1 (*f x*) (*g x*)
shows *Rel* (*rel_fun R*0 *R*1) *f g*
⟨*proof*⟩

lemma *crel_vs_return_app_return*:
assumes *Rel R* (*f x*) (*g x*)
shows *Rel R* (*App* (*Wrap f*) (*Wrap x*)) (⟨*g*⟩ . ⟨*x*⟩)
⟨*proof*⟩

thm *option.case_cong*[*no_vars*]
lemma *option_case_cong*′:
Rel (=) *option*′ *option* ⟹
(*option* = *None* ⟹ *Rel R f*1 *g*1) ⟹
(⋀*x*2. *option* = *Some x*2 ⟹ *Rel R* (*f*2 *x*2) (*g*2 *x*2)) ⟹
Rel R (*case option*′ *of None* ⇒ *f*1 | *Some x*2 ⇒ *f*2 *x*2)

(*case option of None* ⇒ *g*1 | *Some x*2 ⇒ *g*2 *x*2)
⟨*proof*⟩

thm *prod.case_cong*[*no_vars*]
lemma *prod_case_cong'*: **fixes** *prod prod'* **shows**
Rel (=) *prod prod'* ⟹
(⋀*x*1 *x*2. *prod'* = (*x*1, *x*2) ⟹ *Rel R* (*f x*1 *x*2) (*g x*1 *x*2)) ⟹
Rel R (*case prod of* (*x*1, *x*2) ⇒ *f x*1 *x*2)
(*case prod' of* (*x*1, *x*2) ⇒ *g x*1 *x*2)
⟨*proof*⟩

lemmas [*dp_match_rule*] = *prod_case_cong' option_case_cong'*

lemmas [*dp_match_rule*] =
crel_vs_return_app_return

lemmas [*dp_match_rule*] =
map_T_*cong*2
$fold_T$_*cong*2
if_T_*cong*2

lemmas [*dp_match_rule*] =
crel_vs_return
crel_vs_fun_app
*refl*2
*rel_fun*2

end
end

2.7.1 More Heap

lemma *execute_heap_ofD*:
heap_of c h = *h'* **if** *execute c h* = *Some* (*v*, *h'*)
⟨*proof*⟩

lemma *execute_result_ofD*:
result_of c h = *v* **if** *execute c h* = *Some* (*v*, *h'*)
⟨*proof*⟩

locale *heap_correct_init_defs* =

fixes $P :: 'm \Rightarrow heap \Rightarrow bool$
and $lookup :: 'm \Rightarrow 'k \Rightarrow 'v\ option\ Heap$
and $update :: 'm \Rightarrow 'k \Rightarrow 'v \Rightarrow unit\ Heap$
begin

definition map_of_heap' **where**
$map\_of\_heap'\ m\ heap\ k = fst\ (the\ (execute\ (lookup\ m\ k)\ heap))$

end

locale $heap\_correct\_init\_inv = heap\_correct\_init\_defs +$
assumes $lookup\_inv$: $\bigwedge\ m.\ lift\_p\ (P\ m)\ (lookup\ m\ k)$
assumes $update\_inv$: $\bigwedge\ m.\ lift\_p\ (P\ m)\ (update\ m\ k\ v)$

locale $heap\_correct\_init =$
$heap\_correct\_init\_inv +$
assumes $lookup\_correct$:
$\bigwedge\ a.\ P\ a\ m \Longrightarrow map\_of\_heap'\ a\ (snd\ (the\ (execute\ (lookup\ a\ k)\ m)))$
$\subseteq_m\ (map\_of\_heap'\ a\ m)$
and $update\_correct$:
$\bigwedge\ a.\ P\ a\ m \Longrightarrow$
$map\_of\_heap'\ a\ (snd\ (the\ (execute\ (update\ a\ k\ v)\ m))) \subseteq_m (map\_of\_heap'$
$a\ m)(k \mapsto v)$
begin

end

locale $dp\_consistency\_heap\_init = heap\_correct\_init\ \_\ lookup$ **for** $lookup$
$:: 'm \Rightarrow 'k \Rightarrow 'v\ option\ Heap\ +$
fixes $dp :: 'k \Rightarrow 'v$
fixes $init :: 'm\ Heap$
assumes $success$: $success\ init\ Heap.empty$
assumes $empty\_correct$:
$\bigwedge\ empty\ heap.\ execute\ init\ Heap.empty = Some\ (empty,\ heap) \Longrightarrow$
$map\_of\_heap'\ empty\ heap \subseteq_m Map.empty$
and P_empty: $\bigwedge\ empty\ heap.\ execute\ init\ Heap.empty = Some\ (empty,$
$heap) \Longrightarrow P\ empty\ heap$
begin

definition $init\_mem = result\_of\ init\ Heap.empty$

sublocale $dp\_consistency\_heap$
where $P{=}P\ init\_mem$
and $lookup{=}lookup\ init\_mem$

and *update=update init_mem*
⟨*proof*⟩

interpretation *consistent*: *dp_consistency_heap_empty*
where *P=P init_mem*
and *lookup=lookup init_mem*
and *update=update init_mem*
and *empty= heap_of init Heap.empty*
⟨*proof*⟩

lemma *memoized_empty*:
dp x = result_of (*init* ⤜ (λ*mem*. dp_T *mem x*)) *Heap.empty*
if *consistentDP* (dp_T (*result_of init Heap.empty*))
⟨*proof*⟩

end

locale *dp_consistency_heap_init′ = heap_correct_init _ lookup* **for** *lookup* :: *′m* ⇒ *′k* ⇒ *′v option Heap* +
fixes *dp* :: *′k* ⇒ *′v*
fixes *init* :: *′m Heap*
assumes *success*: *success init Heap.empty*
assumes *empty_correct*:
⋀ *empty heap. execute init Heap.empty = Some* (*empty*, *heap*) ⟹ *map_of_heap′ empty heap* ⊆$_m$ *Map.empty*
and *P_empty*: ⋀ *empty heap. execute init Heap.empty = Some* (*empty*, *heap*) ⟹ *P empty heap*
begin

sublocale *dp_consistency_heap*
where *P=P init_mem*
and *lookup=lookup init_mem*
and *update=update init_mem*
⟨*proof*⟩

definition *init_mem = result_of init Heap.empty*

interpretation *consistent*: *dp_consistency_heap_empty*
where *P=P init_mem*
and *lookup=lookup init_mem*
and *update=update init_mem*
and *empty= heap_of init Heap.empty*
⟨*proof*⟩

lemma *memoized_empty*:
dp x = result_of (init ⤜ (λmem. dp$_T$ mem x)) Heap.empty
if *consistentDP init_mem (dp$_T$ (result_of init Heap.empty))*
⟨*proof*⟩

end

locale *dp_consistency_new* =
fixes *dp* :: *'k ⇒ 'v*
fixes *P* :: *'m ⇒ heap ⇒ bool*
and *lookup* :: *'m ⇒ 'k ⇒ 'v option Heap*
and *update* :: *'m ⇒ 'k ⇒ 'v ⇒ unit Heap*
and *init*
assumes
success: *success init Heap.empty*
assumes
inv_init: ⋀ *empty heap. execute init Heap.empty = Some (empty, heap)*
⟹ P empty heap
assumes *consistent*:
⋀ *empty heap. execute init Heap.empty = Some (empty, heap)*
⟹ dp_consistency_heap_empty (P empty) (update empty) (lookup empty) heap
begin

sublocale *dp_consistency_heap_empty*
where *P=P (result_of init Heap.empty)*
and *lookup=lookup (result_of init Heap.empty)*
and *update=update (result_of init Heap.empty)*
and *empty= heap_of init Heap.empty*
⟨*proof*⟩

lemma *memoized_empty*:
dp x = result_of (init ⤜ (λmem. dp$_T$ mem x)) Heap.empty
if *consistentDP (dp$_T$ (result_of init Heap.empty))*
⟨*proof*⟩

end

locale *dp_consistency_new′* =
fixes *dp* :: *'k ⇒ 'v*
fixes *P* :: *'m ⇒ heap ⇒ bool*
and *lookup* :: *'m ⇒ 'k ⇒ 'v option Heap*
and *update* :: *'m ⇒ 'k ⇒ 'v ⇒ unit Heap*
and *init*

and *mem* :: *'m*
assumes *mem_is_init*: *mem* = *result_of init Heap.empty*
assumes
success: *success init Heap.empty*
assumes
inv_init: ⋀ *empty heap. execute init Heap.empty* = *Some* (*empty*, *heap*)
⟹ *P empty heap*
assumes *consistent*:
⋀ *empty heap. execute init Heap.empty* = *Some* (*empty*, *heap*)
⟹ *dp_consistency_heap_empty* (*P empty*) (*update empty*) (*lookup empty*) *heap*
begin

sublocale *dp_consistency_heap_empty*
where *P*=*P mem*
and *lookup*=*lookup mem*
and *update*=*update mem*
and *empty*= *heap_of init Heap.empty*
⟨*proof*⟩

lemma *memoized_empty*:
dp x = *result_of* (*init* ⤜ (λ*mem. dp*$_T$ *mem x*)) *Heap.empty*
if *consistentDP* (*dp*$_T$ (*result_of init Heap.empty*))
⟨*proof*⟩

end

locale *dp_consistency_heap_array_new'* =
fixes *size* :: *nat*
and *to_index* :: (*'k* :: *heap*) ⇒ *nat*
and *mem* :: (*'v*::*heap*) *option array*
and *dp* :: *'k* ⇒ *'v*::*heap*
assumes *mem_is_init*: *mem* = *result_of* (*mem_empty size*) *Heap.empty*
assumes *injective*: *injective size to_index*
begin

sublocale *dp_consistency_new'*
where *P* = λ *mem heap. Array.length heap mem* = *size*
and *lookup* = λ *mem. mem_lookup size to_index mem*
and *update* = λ *mem. mem_update size to_index mem*
and *init* = *mem_empty size*
and *mem* = *mem*
⟨*proof*⟩

thm *memoized_empty*

end

locale *dp_consistency_heap_array_new* =
  fixes *size* :: *nat*
    and *to_index* :: (*'k* :: *heap*) ⇒ *nat*
    and *dp* :: *'k* ⇒ *'v*::*heap*
  assumes *injective*: *injective size to_index*
begin

sublocale *dp_consistency_new*
  where *P* = λ *mem heap. Array.length heap mem = size*
    and *lookup* = λ *mem. mem_lookup size to_index mem*
    and *update* = λ *mem. mem_update size to_index mem*
    and *init* = *mem_empty size*
  ⟨*proof*⟩

thm *memoized_empty*

end

locale *dp_consistency_heap_array* =
  fixes *size* :: *nat*
    and *to_index* :: (*'k* :: *heap*) ⇒ *nat*
    and *dp* :: *'k* ⇒ *'v*::*heap*
  assumes *injective*: *injective size to_index*
begin

sublocale *dp_consistency_heap_init*
  where *P*=λ*mem heap. Array.length heap mem = size*
    and *lookup*=λ *mem. mem_lookup size to_index mem*
    and *update*=λ *mem. mem_update size to_index mem*
    and *init*=*mem_empty size*
  ⟨*proof*⟩

end

locale *dp_consistency_heap_array_pair'* =
  fixes *size* :: *nat*
  fixes *key1* :: *'k* ⇒ (*'k1* :: *heap*) **and** *key2* :: *'k* ⇒ *'k2* :: *heap*
    and *to_index* :: *'k2* ⇒ *nat*
    and *dp* :: *'k* ⇒ *'v*::*heap*

and $k1\ k2 :: 'k1$
and $mem :: ('k1\ ref \times$
$'k1\ ref \times$
$'v\ option\ array\ ref \times$
$'v\ option\ array\ ref)$
assumes *mem_is_init*: $mem = result\_of\ (init\_state\ size\ k1\ k2)\ Heap.empty$
assumes *injective*: $injective\ size\ to\_index$
and *keys_injective*: $\forall\, k\ k'.\ key1\ k = key1\ k' \wedge key2\ k = key2\ k' \longrightarrow k = k'$
and *keys_neq*: $k1 \neq k2$
begin

definition
$inv\_pair' = (\lambda\ (k\_ref1,\ k\_ref2,\ m\_ref1,\ m\_ref2).$
$pair\_mem\_defs.inv\_pair\ (lookup1\ size\ to\_index\ m\_ref1)$
$(lookup2\ size\ to\_index\ m\_ref2)\ (get\_k1\ k\_ref1)$
$(get\_k2\ k\_ref2)$
$(inv\_pair\_weak\ size\ m\_ref1\ m\_ref2\ k\_ref1\ k\_ref2)\ key1\ key2)$

sublocale $dp\_consistency\_new'$
where $P{=}inv\_pair'$
and $lookup{=}\lambda\ (k\_ref1,\ k\_ref2,\ m\_ref1,\ m\_ref2).$
$lookup\_pair\ size\ to\_index\ key1\ key2\ m\_ref1\ m\_ref2\ k\_ref1\ k\_ref2$
and $update{=}\lambda\ (k\_ref1,\ k\_ref2,\ m\_ref1,\ m\_ref2).$
$update\_pair\ size\ to\_index\ key1\ key2\ m\_ref1\ m\_ref2\ k\_ref1\ k\_ref2$
and $init{=}init\_state\ size\ k1\ k2$
⟨*proof*⟩

end

locale $dp\_consistency\_heap\_array\_pair\_iterator =$
$dp\_consistency\_heap\_array\_pair'$ **where** $dp = dp + iterator$ **where** $cnt = cnt$
for $dp :: 'k \Rightarrow 'v{::}heap$ **and** $cnt :: 'k \Rightarrow bool$
begin

sublocale $dp\_consistency\_iterator\_heap$
where $P = inv\_pair'\ mem$
and $update = (case\ mem\ of$
$(k\_ref1,\ k\_ref2,\ m\_ref1,\ m\_ref2) \Rightarrow$
$update\_pair\ size\ to\_index\ key1\ key2\ m\_ref1\ m\_ref2\ k\_ref1\ k\_ref2)$
and $lookup = (case\ mem\ of$
$(k\_ref1,\ k\_ref2,\ m\_ref1,\ m\_ref2) \Rightarrow$
$lookup\_pair\ size\ to\_index\ key1\ key2\ m\_ref1\ m\_ref2\ k\_ref1\ k\_ref2)$

⟨*proof*⟩

end

locale *dp_consistency_heap_array_pair* =
fixes *size* :: *nat*
fixes *key*1 :: ′*k* ⇒ (′*k*1 :: *heap*) **and** *key*2 :: ′*k* ⇒ ′*k*2 :: *heap*
and *to_index* :: ′*k*2 ⇒ *nat*
and *dp* :: ′*k* ⇒ ′*v*::*heap*
and *k*1 *k*2 :: ′*k*1
assumes *injective*: *injective size to_index*
and *keys_injective*: ∀ *k k*′. *key*1 *k* = *key*1 *k*′ ∧ *key*2 *k* = *key*2 *k*′ ⟶ *k* = *k*′
and *keys_neq*: *k*1 ≠ *k*2
begin

definition
inv_pair′ = (λ (*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2).
pair_mem_defs.inv_pair (*lookup*1 *size to_index m_ref*1)
(*lookup*2 *size to_index m_ref*2) (*get_k*1 *k_ref*1)
(*get_k*2 *k_ref*2)
(*inv_pair_weak size m_ref*1 *m_ref*2 *k_ref*1 *k_ref*2) *key*1 *key*2)

sublocale *dp_consistency_new*
where *P*=*inv_pair*′
and *lookup*=λ (*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2).
*lookup_pair size to_index key*1 *key*2 *m_ref*1 *m_ref*2 *k_ref*1 *k_ref*2
and *update*=λ (*k_ref*1, *k_ref*2, *m_ref*1, *m_ref*2).
*update_pair size to_index key*1 *key*2 *m_ref*1 *m_ref*2 *k_ref*1 *k_ref*2
and *init*=*init_state size k*1 *k*2
⟨*proof*⟩

end

2.7.2 Code Setup

lemmas [*code_unfold*] = *heap_mem_defs.checkmem_checkmem*′[*symmetric*]
lemmas [*code*] =
heap_mem_defs.checkmem′*_def*
Heap_Main.map$_T$*_def*

end

2.8 Setup for the State Monad

theory *State_Main*
imports
../transform/Transform_Cmd
Memory
begin

context includes *state_monad_syntax* **begin**

thm *if_cong*
lemma *ifT_cong*:
assumes $b = c$ $c \Longrightarrow x = u$ $\neg c \Longrightarrow y = v$
shows *State_Monad_Ext.if*$_T$ $\langle b \rangle$ x y = *State_Monad_Ext.if*$_T$ $\langle c \rangle$ u v
⟨*proof*⟩

lemma *return_app_return_cong*:
assumes $f\ x = g\ y$
shows $\langle f \rangle$. $\langle x \rangle$ = $\langle g \rangle$. $\langle y \rangle$
⟨*proof*⟩

lemmas [*fundef_cong*] =
return_app_return_cong
ifT_cong
end

memoize_fun *comp*$_T$: *comp* **monadifies** (*state*) *comp_def*
lemma (**in** *dp_consistency*) *comp*$_T$*_transfer*[*transfer_rule*]:
crel_vs (($R1$ ===>$_T$ $R2$) ===>$_T$ ($R0$ ===>$_T$ $R1$) ===>$_T$ ($R0$ ===>$_T$ $R2$)) *comp* *comp*$_T$
⟨*proof*⟩

memoize_fun *map*$_T$: *map* **monadifies** (*state*) *list.map*
lemma (**in** *dp_consistency*) *map*$_T$*_transfer*[*transfer_rule*]:
crel_vs (($R0$ ===>$_T$ $R1$) ===>$_T$ *list_all2* $R0$ ===>$_T$ *list_all2* $R1$) *map* *map*$_T$
⟨*proof*⟩

memoize_fun *fold*$_T$: *fold* **monadifies** (*state*) *fold.simps*
lemma (**in** *dp_consistency*) *fold*$_T$*_transfer*[*transfer_rule*]:
crel_vs (($R0$ ===>$_T$ $R1$ ===>$_T$ $R1$) ===>$_T$ *list_all2* $R0$ ===>$_T$ $R1$ ===>$_T$ $R1$) *fold* *fold*$_T$
⟨*proof*⟩

context includes *state_monad_syntax* **begin**

thm *map_cong*
lemma *mapT_cong*:
assumes $xs = ys$ $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
shows $map_T\ .\ \langle f \rangle\ .\ \langle xs \rangle = map_T\ .\ \langle g \rangle\ .\ \langle ys \rangle$
⟨*proof*⟩

thm *fold_cong*
lemma *foldT_cong*:
assumes $xs = ys$ $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
shows $fold_T\ .\ \langle f \rangle\ .\ \langle xs \rangle = fold_T\ .\ \langle g \rangle\ .\ \langle ys \rangle$
⟨*proof*⟩

lemma *abs_unit_cong*:

assumes $x = y$
shows $(\lambda\_::unit.\ x) = (\lambda\_.\ y)$
⟨*proof*⟩

lemmas [*fundef_cong*] =
return_app_return_cong
ifT_cong
mapT_cong
foldT_cong
abs_unit_cong
end

context *dp_consistency* **begin**
context includes *lifting_syntax state_monad_syntax* **begin**

named_theorems *dp_match_rule*

thm *if_cong*
lemma *if*$_T$*_cong*2:
assumes *Rel* (=) *b c c* ⟹ *Rel* (*crel_vs R*) *x x*$_T$ ¬*c* ⟹ *Rel* (*crel_vs R*) *y y*$_T$
shows *Rel* (*crel_vs R*) (*if* (*Wrap b*) *then x else y*) (*State_Monad_Ext.if*$_T$ ⟨*c*⟩ *x*$_T$ *y*$_T$)
⟨*proof*⟩

lemma *map*$_T$*_cong*2:
assumes
is_equality R

Rel R xs ys
⋀x. x∈set ys ⟹ Rel (crel_vs S) (f x) (f_T' x)
shows *Rel (crel_vs (list_all2 S)) (App (App map (Wrap f)) (Wrap xs))*
(map_T . ⟨f_T'⟩ . ⟨ys⟩)
⟨*proof*⟩

lemma *fold_T_cong2*:
assumes
is_equality R
Rel R xs ys
⋀x. x∈set ys ⟹ Rel (crel_vs (S ===> crel_vs S)) (f x) (f_T' x)
shows
Rel (crel_vs (S ===> crel_vs S)) (fold f xs) (fold_T . ⟨f_T'⟩ . ⟨ys⟩)
⟨*proof*⟩

lemma *refl2*:
is_equality R ⟹ Rel R x x
⟨*proof*⟩

lemma *rel_fun2*:
assumes *is_equality R0 ⋀x. Rel R1 (f x) (g x)*
shows *Rel (rel_fun R0 R1) f g*
⟨*proof*⟩

lemma *crel_vs_return_app_return*:
assumes *Rel R (f x) (g x)*
shows *Rel R (App (Wrap f) (Wrap x)) (⟨g⟩ . ⟨x⟩)*
⟨*proof*⟩

thm *option.case_cong[no_vars]*
lemma *option_case_cong'*:
Rel (=) option' option ⟹
(option = None ⟹ Rel R f1 g1) ⟹
(⋀x2. option = Some x2 ⟹ Rel R (f2 x2) (g2 x2)) ⟹
Rel R (case option' of None ⇒ f1 | Some x2 ⇒ f2 x2)
(case option of None ⇒ g1 | Some x2 ⇒ g2 x2)
⟨*proof*⟩

thm *prod.case_cong[no_vars]*
lemma *prod_case_cong'*: **fixes** *prod prod'* **shows**
Rel (=) prod prod' ⟹
(⋀x1 x2. prod' = (x1, x2) ⟹ Rel R (f x1 x2) (g x1 x2)) ⟹
Rel R (case prod of (x1, x2) ⇒ f x1 x2)
(case prod' of (x1, x2) ⇒ g x1 x2)

⟨*proof*⟩

thm *nat.case_cong*[*no_vars*]
lemma *nat_case_cong'*: **fixes** *nat nat'* **shows**
Rel (=) *nat nat'* ⟹
(*nat'* = 0 ⟹ *Rel R f*1 *g*1) ⟹
(⋀*x*2. *nat'* = *Suc x*2 ⟹ *Rel R* (*f*2 *x*2) (*g*2 *x*2)) ⟹
Rel R (*case nat of* 0 ⇒ *f*1 | *Suc x*2 ⇒ *f*2 *x*2) (*case nat' of* 0 ⇒ *g*1 | *Suc x*2 ⇒ *g*2 *x*2)
⟨*proof*⟩

lemmas [*dp_match_rule*] =
prod_case_cong'
option_case_cong'
nat_case_cong'

lemmas [*dp_match_rule*] =
crel_vs_return_app_return

lemmas [*dp_match_rule*] =
map$_T$*_cong*2
fold$_T$*_cong*2
if$_T$*_cong*2

lemmas [*dp_match_rule*] =
crel_vs_return
crel_vs_fun_app
*refl*2
*rel_fun*2

end
end

2.8.1 Code Setup

lemmas [*code_unfold*] =
state_mem_defs.checkmem_checkmem'[*symmetric*]
state_mem_defs.checkmem'_def
map$_T$*_def*

end

3 Examples

3.1 Misc

theory *Example_Misc*
imports
Main
HOL−Library.Extended
../state_monad/State_Main
begin

Lists **fun** *min_list* :: *'a::ord list* ⇒ *'a* **where**
min_list (*x* # *xs*) = (*case xs of* [] ⇒ *x* | _ ⇒ *min x* (*min_list xs*))

lemma *fold_min_commute*:
fold min xs (*min a b*) = *min a* (*fold min xs b*) **for** *a* :: *'a* :: *linorder*
⟨*proof*⟩

lemma *min_list_fold*:
min_list (*x* # *xs*) = *fold min xs x* **for** *x* :: *'a* :: *linorder*
⟨*proof*⟩

lemma *induct_list*012:
⟦*P* []; ⋀*x*. *P* [*x*]; ⋀*x y zs*. *P* (*y* # *zs*) ⟹ *P* (*x* # *y* # *zs*)⟧ ⟹ *P xs*
⟨*proof*⟩

lemma *min_list_Min*: *xs* ≠ [] ⟹ *min_list xs* = *Min* (*set xs*)
⟨*proof*⟩

Extended Data Type **lemma** *Pinf_add_right*[*simp*]:
∞ + *x* = ∞
⟨*proof*⟩

Syntax **bundle** *app_syntax* **begin**

notation *App* (**infixl** $ 999)
notation *Wrap* (⟪_⟫)

end

end
theory *Tracing*
imports
../heap_monad/Heap_Main
HOL−Library.Code_Target_Numeral
Show.Show_Instances
begin

NB: A more complete solution could be built by using the following entry: `https://www.isa-afp.org/entries/Show.html`.

definition *writeln* :: *String.literal* ⇒ *unit* **where**
writeln = (λ *s*. ())

code_printing
constant *writeln* ⇀ (*SML*) *writeln* _

definition *trace* **where**
trace s x = (*let a* = *writeln s in x*)

lemma *trace_alt_def*[*simp*]:
trace s x = (λ _. *x*) (*writeln s*)
⟨*proof*⟩

definition (**in** *heap_mem_defs*) *checkmem_trace* ::
('*k* ⇒ *String.literal*) ⇒ '*k* ⇒ (*unit* ⇒ '*v Heap*) ⇒ '*v Heap*
where
checkmem_trace trace_key param calc ≡
Heap_Monad.bind (*lookup param*) (λ *x*.
case x of
Some x ⇒ *trace* (*STR ''Hit ''* + *trace_key param*) (*return x*)
| *None* ⇒ *trace* (*STR ''Miss ''* + *trace_key param*)
Heap_Monad.bind (*calc* ()) (λ *x*.
Heap_Monad.bind (*update param x*) (λ _.
return x
)
)
)

lemma (**in** *heap_mem_defs*) *checkmem_checkmem_trace*:
checkmem param calc = *checkmem_trace trace_key param* (λ_. *calc*)
⟨*proof*⟩

definition *nat_to_string* :: *nat* ⇒ *String.literal* **where**
nat_to_string x = *String.implode* (*show x*)

definition *nat_pair_to_string* :: *nat* × *nat* ⇒ *String.literal* **where**
nat_pair_to_string x = *String.implode* (*show x*)

value *show* (3 :: *nat*)

Code Setup **lemmas** [*code*] =
heap_mem_defs.checkmem_trace_def

lemmas [*code_unfold*] =
heap_mem_defs.checkmem_checkmem_trace[**where** *trace_key* = *nat_to_string*]
heap_mem_defs.checkmem_checkmem_trace[**where** *trace_key* = *nat_pair_to_string*]

end
theory *Ground_Function*
imports *Main*
keywords
ground_function :: *thy_decl*
begin

⟨*ML*⟩

end

3.2 The Bellman-Ford Algorithm

theory *Bellman_Ford*
imports
HOL−Library.IArray
HOL−Library.Code_Target_Numeral
HOL−Library.Product_Lexorder
HOL−Library.RBT_Mapping
../heap_monad/Heap_Main
Example_Misc
../util/Tracing
../util/Ground_Function
begin

3.2.1 Misc

lemma *nat_le_cases*:
fixes *n* :: *nat*

assumes $i \leq n$
obtains $i < n \mid i = n$
⟨*proof*⟩

context *dp_consistency_iterator*
begin

lemma *crel_vs_iterate_state*:
crel_vs (=) () (*iter_state f x*) **if** ((=) ===>$_T$ *R*) *g f*
⟨*proof*⟩

lemma *consistent_crel_vs_iterate_state*:
crel_vs (=) () (*iter_state f x*) **if** *consistentDP f*
⟨*proof*⟩

end

instance *extended* :: (*countable*) *countable*
⟨*proof*⟩

instance *extended* :: (*heap*) *heap* ⟨*proof*⟩

instantiation *extended* :: (*conditionally_complete_lattice*) *complete_lattice*
begin

definition
Inf A = (
if A = {} ∨ *A* = {∞} *then* ∞
else if −∞ ∈ *A* ∨ ¬ *bdd_below* (*Fin* −‘ *A*) *then* −∞
else Fin (*Inf* (*Fin* −‘ *A*)))

definition
Sup A = (
if A = {} ∨ *A* = {−∞} *then* −∞
else if ∞ ∈ *A* ∨ ¬ *bdd_above* (*Fin* −‘ *A*) *then* ∞
else Fin (*Sup* (*Fin* −‘ *A*)))

instance
⟨*proof*⟩

end

instance *extended* :: ({*conditionally_complete_lattice*,*linorder*}) *complete_linorder*
⟨*proof*⟩

lemma *Minf_eq_zero*[*simp*]: $-\infty = 0 \longleftrightarrow$ *False* **and** *Pinf_eq_zero*[*simp*]: $\infty = 0 \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *Sup_int*:
fixes x :: *int* **and** X :: *int set*
assumes $X \neq \{\}$ *bdd_above* X
shows *Sup* $X \in X \wedge (\forall y{\in}X.\ y \leq Sup\ X)$
⟨*proof*⟩

lemmas *Sup_int_in* = *Sup_int*[*THEN conjunct1*]

lemma *Inf_int_in*:
fixes S :: *int set*
assumes $S \neq \{\}$ *bdd_below* S
shows *Inf* $S \in S$
⟨*proof*⟩

lemma *finite_setcompr_eq_image*: *finite* $\{f\ x\ |x.\ P\ x\} \longleftrightarrow$ *finite* $(f\ `\ \{x.\ P\ x\})$
⟨*proof*⟩

lemma *finite_lists_length_le*1: *finite* $\{xs.\ length\ xs \leq i \wedge set\ xs \subseteq \{0..(n::nat)\}\}$ **for** i
⟨*proof*⟩

lemma *finite_lists_length_le*2: *finite* $\{xs.\ length\ xs + 1 \leq i \wedge set\ xs \subseteq \{0..(n::nat)\}\}$ **for** i
⟨*proof*⟩

lemmas [*simp*] =
*finite_setcompr_eq_image finite_lists_length_le*2[*simplified*] *finite_lists_length_le*1

lemma *get_return*:
run_state (*State_Monad.bind State_Monad.get* (λ *m*. *State_Monad.return* $(f\ m)))\ m = (f\ m,\ m)$
⟨*proof*⟩

lemma *list_pidgeonhole*:

assumes *set xs* $\subseteq$ *S card S* $<$ *length xs finite S*
obtains *as a bs cs* **where** *xs* = *as* @ *a* # *bs* @ *a* # *cs*
⟨*proof*⟩

lemma *path_eq_cycleE*:
assumes *v* # *ys* @ [*t*] = *as* @ *a* # *bs* @ *a* # *cs*
obtains (*Nil_Nil*) *as* = [] *cs* = [] *v* = *a a* = *t ys* = *bs*
| (*Nil_Cons*) *cs*′ **where** *as* = [] *v* = *a ys* = *bs* @ *a* # *cs*′ *cs* = *cs*′ @ [*t*]
| (*Cons_Nil*) *as*′ **where** *as* = *v* # *as*′ *cs* = [] *a* = *t ys* = *as*′ @ *a* # *bs*
| (*Cons_Cons*) *as*′ *cs*′ **where** *as* = *v* # *as*′ *cs* = *cs*′ @ [*t*] *ys* = *as*′ @ *a* # *bs* @ *a* # *cs*′
⟨*proof*⟩

lemma *le_add_same_cancel*1:
a + *b* ≥ *a* ⟷ *b* ≥ 0 **if** *a* < ∞ −∞ < *a* **for** *a b* :: *int extended*
⟨*proof*⟩

lemma *add_gt_minfI*:
assumes −∞ < *a* −∞ < *b*
shows −∞ < *a* + *b*
⟨*proof*⟩

lemma *add_lt_infI*:
assumes *a* < ∞ *b* < ∞
shows *a* + *b* < ∞
⟨*proof*⟩

lemma *sum_list_not_infI*:
sum_list xs < ∞ **if** ∀ *x* ∈ *set xs*. *x* < ∞ **for** *xs* :: *int extended list*
⟨*proof*⟩

lemma *sum_list_not_minfI*:
sum_list xs > −∞ **if** ∀ *x* ∈ *set xs*. *x* > −∞ **for** *xs* :: *int extended list*
⟨*proof*⟩

3.2.2 Single-Sink Shortest Path Problem

datatype *bf_result* = *Path nat list int* | *No_Path* | *Computation_Error*

context
fixes *n* :: *nat* **and** *W* :: *nat* ⇒ *nat* ⇒ *int extended*
begin

context

fixes $t :: nat$ — Final node
begin

The correctness proof closely follows Kleinberg & Tardos: "Algorithm Design", chapter "Dynamic Programming" [1]

fun *weight* :: *nat list* ⇒ *int extended* **where**
weight [*v*] = 0
| *weight* (*v* # *w* # *xs*) = *W v w* + *weight* (*w* # *xs*)

definition
OPT i v = (
Min (
{*weight* (*v* # *xs* @ [*t*]) | *xs*. *length xs* + 1 ≤ *i* ∧ *set xs* ⊆ {0..*n*}} ∪
{*if t* = *v then* 0 *else* ∞}
)
)

lemma *weight_alt_def'*:
weight (*s* # *xs*) + *w* = *snd* (*fold* (λ*j* (*i*, *x*). (*j*, *W i j* + *x*)) *xs* (*s*, *w*))
⟨*proof*⟩

lemma *weight_alt_def*:
weight (*s* # *xs*) = *snd* (*fold* (λ*j* (*i*, *x*). (*j*, *W i j* + *x*)) *xs* (*s*, 0))
⟨*proof*⟩

lemma *weight_append*:
weight (*xs* @ *a* # *ys*) = *weight* (*xs* @ [*a*]) + *weight* (*a* # *ys*)
⟨*proof*⟩

lemma *OPT_*0:
OPT 0 *v* = (*if t* = *v then* 0 *else* ∞)
⟨*proof*⟩

3.2.3 Functional Correctness

lemma *OPT_cases*:
obtains (*path*) *xs* **where** *OPT i v* = *weight* (*v* # *xs* @ [*t*]) *length xs* + 1 ≤ *i set xs* ⊆ {0..*n*}
| (*sink*) *v* = *t OPT i v* = 0
| (*unreachable*) *v* ≠ *t OPT i v* = ∞
⟨*proof*⟩

lemma *OPT_Suc*:
OPT (*Suc i*) *v* = *min* (*OPT i v*) (*Min* {*OPT i w* + *W v w* | *w*. *w* ≤ *n*})

(**is** *?lhs = ?rhs*)
  if $t \leq n$
⟨*proof*⟩

fun *bf* :: *nat* ⇒ *nat* ⇒ *int extended* **where**
  bf 0 *v* = (*if t = v then* 0 *else* ∞)
| *bf* (*Suc i*) *v* = *min_list*
      (*bf i v* # [*W v w* + *bf i w* . *w* ← [0 ..< *Suc n*]])

lemmas [*simp del*] = *bf.simps*
lemmas *bf_simps*[*simp*] = *bf.simps*[*unfolded min_list_fold*]

lemma *bf_correct*:
  OPT i j = *bf i j* **if** ‹$t \leq n$›
⟨*proof*⟩

3.2.4 Functional Memoization

memoize_fun bf_m: *bf* **with_memory** *dp_consistency_mapping* **monadifies** (*state*) *bf.simps*

Generated Definitions

context includes *state_monad_syntax* **begin**
thm bf_m'*.simps* bf_m*_def*
end

Correspondence Proof

memoize_correct
  ⟨*proof*⟩
print_theorems
lemmas [*code*] = bf_m*.memoized_correct*

interpretation *iterator*
  λ (*x*, *y*). $x \leq n \wedge y \leq n$
  λ (*x*, *y*). *if* $y < n$ *then* (x, $y + 1$) *else* ($x + 1$, 0)
  λ (*x*, *y*). $x * (n + 1) + y$
  ⟨*proof*⟩

interpretation *bottom_up*: *dp_consistency_iterator_empty*
  λ (_::(*nat* × *nat*, *int extended*) *mapping*). *True*
  λ (*x*, *y*). *bf x y*
  λ *k*. *do* {*m* ← *State_Monad.get*; *State_Monad.return* (*Mapping.lookup m k* :: *int extended option*)}

```
  λ k v. do {m ← State_Monad.get; State_Monad.set (Mapping.update k v
m)}
  λ (x, y). x ≤ n ∧ y ≤ n
  λ (x, y). if y < n then (x, y + 1) else (x + 1, 0)
  λ (x, y). x * (n + 1) + y
  Mapping.empty ⟨proof⟩

definition
  iter_bf = iter_state (λ (x, y). bf_m' x y)

lemma iter_bf_unfold[code]:
  iter_bf = (λ (i, j).
    (if i ≤ n ∧ j ≤ n
     then do {
            bf_m' i j;
            iter_bf (if j < n then (i, j + 1) else (i + 1, 0))
          }
     else State_Monad.return ()))
  ⟨proof⟩

lemmas bf_memoized = bf_m.memoized[OF bf_m.crel]
lemmas bf_bottom_up = bottom_up.memoized[OF bf_m.crel, folded iter_bf_def]
```

This will be our final implementation, which includes detection of negative cycles. See the corresponding section below for the correctness proof.

```
definition
  bellman_ford ≡
    do {
      _  ← iter_bf (n, n);
      xs ← State_Main.map_T' (λi. bf_m' n i) [0..<n+1];
      ys ← State_Main.map_T' (λi. bf_m' (n + 1) i) [0..<n+1];
      State_Monad.return (if xs = ys then Some xs else None)
    }

context
  includes state_monad_syntax
begin

lemma bellman_ford_alt_def:
  bellman_ford ≡
    do {
      _  ← iter_bf (n, n);
      (⟨λxs. ⟨λys. State_Monad.return (if xs = ys then Some xs else None)⟩
      . (State_Main.map_T . ⟨λi. bf_m' (n + 1) i⟩ . ⟨[0..<n+1]⟩)⟩)
```

. (*State_Main.map*$_T$. ⟨λ*i*. *bf*$_m$′ *n i*⟩ . ⟨[0..<*n*+1]⟩)
}
⟨*proof*⟩

end

3.2.5 Imperative Memoization

context
fixes *mem* :: *nat ref* × *nat ref* × *int extended option array ref* × *int extended option array ref*
assumes *mem_is_init*: *mem* = *result_of* (*init_state* (*n* + 1) 1 0) *Heap.empty*
begin

lemma [*intro*]:
dp_consistency_heap_array_pair′ (*n* + 1) *fst snd id* 1 0 *mem*
⟨*proof*⟩

interpretation *iterator*
λ (*x*, *y*). *x* ≤ *n* ∧ *y* ≤ *n*
λ (*x*, *y*). *if y* < *n then* (*x*, *y* + 1) *else* (*x* + 1, 0)
λ (*x*, *y*). *x* ∗ (*n* + 1) + *y*
⟨*proof*⟩

lemma [*intro*]:
dp_consistency_heap_array_pair_iterator (*n* + 1) *fst snd id* 1 0 *mem*
(λ (*x*, *y*). *if y* < *n then* (*x*, *y* + 1) *else* (*x* + 1, 0))
(λ (*x*, *y*). *x* ∗ (*n* + 1) + *y*)
(λ (*x*, *y*). *x* ≤ *n* ∧ *y* ≤ *n*)
⟨*proof*⟩

memoize_fun *bf*$_h$: *bf*
with_memory (**default_proof**) *dp_consistency_heap_array_pair_iterator*
where *size* = *n* + 1
and *key*1 = *fst* :: *nat* × *nat* ⇒ *nat* **and** *key*2 = *snd* :: *nat* × *nat* ⇒ *nat*
and *k*1 = 1 :: *nat* **and** *k*2 = 0 :: *nat*
and *to_index* = *id* :: *nat* ⇒ *nat*
and *mem* = *mem*
and *cnt* = λ (*x*, *y*). *x* ≤ *n* ∧ *y* ≤ *n*
and *nxt* = λ (*x* :: *nat*, *y*). *if y* < *n then* (*x*, *y* + 1) *else* (*x* + 1, 0)
and *sizef* = λ (*x*, *y*). *x* ∗ (*n* + 1) + *y*
monadifies (*heap*) *bf.simps*

memoize_correct

⟨*proof*⟩

lemmas *memoized_empty* = bf_h.*memoized_empty*[*OF* bf_h.*consistent_DP_iter_and_compute*[*OF* bf_h.*crel*]]
lemmas *iter_heap_unfold* = *iter_heap_unfold*

end

3.2.6 Detecting Negative Cycles

definition
shortest v = (
Inf (
{*weight* (v # *xs* @ [t]) | *xs*. *set xs* ⊆ {0..n}} ∪
{*if* $t = v$ *then* 0 *else* ∞}
)
)

definition
is_path xs ≡ *weight* (*xs* @ [t]) < ∞

definition
has_negative_cycle ≡
∃ *xs a ys*. *set* (a # *xs* @ *ys*) ⊆ {0..n} ∧ *weight* (a # *xs* @ [a]) < 0 ∧ *is_path* (a # *ys*)

definition
reaches a ≡ ∃ *xs*. *is_path* (a # *xs*) ∧ $a \le n$ ∧ *set xs* ⊆ {0..n}

lemma *fold_sum_aux′*:
assumes ∀ u ∈ *set* (a # *xs*). ∀ v ∈ *set* (*xs* @ [b]). $f\ v + W\ u\ v \ge f\ u$
shows *sum_list* (*map* f (a # *xs*)) ≤ *sum_list* (*map* f (*xs* @ [b])) + *weight* (a # *xs* @ [b])
⟨*proof*⟩

lemma *fold_sum_aux*:
assumes ∀ u ∈ *set* (a # *xs*). ∀ v ∈ *set* (a # *xs*). $f\ v + W\ u\ v \ge f\ u$
shows *sum_list* (*map* f (a # *xs* @ [a])) ≤ *sum_list* (*map* f (a # *xs* @ [a])) + *weight* (a # *xs* @ [a])
⟨*proof*⟩

context
begin

private definition *is_path2 xs* ≡ *weight xs* < ∞

private lemma *is_path2_remove_cycle*:
assumes *is_path2* (*as* @ *a* # *bs* @ *a* # *cs*)
shows *is_path2* (*as* @ *a* # *cs*)
⟨*proof*⟩ **lemma** *is_path_eq*:
is_path xs ⟷ *is_path2* (*xs* @ [*t*])
⟨*proof*⟩

lemma *is_path_remove_cycle*:
assumes *is_path* (*as* @ *a* # *bs* @ *a* # *cs*)
shows *is_path* (*as* @ *a* # *cs*)
⟨*proof*⟩

lemma *is_path_remove_cycle2*:
assumes *is_path* (*as* @ *t* # *cs*)
shows *is_path as*
⟨*proof*⟩

end

lemma *is_path_shorten*:
assumes *is_path* (*i* # *xs*) *i* ≤ *n set xs* ⊆ {0..*n*} *t* ≤ *n t* ≠ *i*
obtains *xs* **where** *is_path* (*i* # *xs*) *i* ≤ *n set xs* ⊆ {0..*n*} *length xs* < *n*
⟨*proof*⟩

lemma *reaches_non_inf_path*:
assumes *reaches i i* ≤ *n t* ≤ *n*
shows *OPT n i* < ∞
⟨*proof*⟩

lemma *OPT_sink_le_*0:
OPT i t ≤ 0
⟨*proof*⟩

lemma *is_path_appendD*:
assumes *is_path* (*as* @ *a* # *bs*)
shows *is_path* (*a* # *bs*)
⟨*proof*⟩

lemma *has_negative_cycleI*:
assumes *set* (*a* # *xs* @ *ys*) ⊆ {0..*n*} *weight* (*a* # *xs* @ [*a*]) < 0 *is_path* (*a* # *ys*)
shows *has_negative_cycle*

⟨*proof*⟩

lemma *OPT_cases*2:
obtains (*path*) *xs* **where**
$v \neq t$ *OPT* i $v \neq \infty$ *OPT* i $v =$ *weight* $(v \# xs @ [t])$ *length* $xs + 1 \leq i$
set $xs \subseteq \{0..n\}$
| (*unreachable*) $v \neq t$ *OPT* i $v = \infty$
| (*sink*) $v = t$ *OPT* i $v \leq 0$
⟨*proof*⟩

lemma *shortest_le_OPT*:
assumes $v \leq n$
shows *shortest* $v \leq$ *OPT* i v
⟨*proof*⟩

context
assumes *W_wellformed*: $\forall\, i \leq n.\ \forall\, j \leq n.\ W\ i\ j > -\infty$
assumes $t \leq n$
begin

lemma *weight_not_minfI*:
$-\infty <$ *weight* xs **if** *set* $xs \subseteq \{0..n\}$ $xs \neq []$
⟨*proof*⟩

lemma *OPT_not_minfI*:
OPT n $i > -\infty$ **if** $i \leq n$
⟨*proof*⟩

theorem *detects_cycle*:
assumes *has_negative_cycle*
shows $\exists\, i \leq n.$ *OPT* $(n + 1)$ $i <$ *OPT* n i
⟨*proof*⟩

corollary *bf_detects_cycle*:
assumes *has_negative_cycle*
shows $\exists\, i \leq n.$ *bf* $(n + 1)$ $i <$ *bf* n i
⟨*proof*⟩

lemma *shortest_cases*:
assumes $v \leq n$
obtains (*path*) *xs* **where** *shortest* $v =$ *weight* $(v \# xs @ [t])$ *set* $xs \subseteq \{0..n\}$
| (*sink*) $v = t$ *shortest* $v = 0$

| (*unreachable*) $v \neq t$ *shortest* $v = \infty$
| (*negative_cycle*) *shortest* $v = -\infty$ $\forall x.$ $\exists xs.$ *set* $xs \subseteq \{0..n\} \wedge$ *weight* $(v \# xs @ [t]) <$ *Fin* x
⟨*proof*⟩

lemma *simple_paths*:
assumes ¬ *has_negative_cycle* *weight* $(v \# xs @ [t]) < \infty$ *set* $xs \subseteq \{0..n\}$ $v \leq n$
obtains *ys* **where**
weight $(v \# ys @ [t]) \leq$ *weight* $(v \# xs @ [t])$ *set* $ys \subseteq \{0..n\}$ *length* $ys < n \mid v = t$
⟨*proof*⟩

theorem *shorter_than_OPT_n_has_negative_cycle*:
assumes *shortest* $v <$ *OPT* n v $v \leq n$
shows *has_negative_cycle*
⟨*proof*⟩

corollary *detects_cycle_has_negative_cycle*:
assumes *OPT* $(n + 1)$ $v <$ *OPT* n v $v \leq n$
shows *has_negative_cycle*
⟨*proof*⟩

corollary *bellman_ford_detects_cycle*:
has_negative_cycle $\longleftrightarrow$ $(\exists v \leq n.$ *OPT* $(n + 1)$ $v <$ *OPT* n $v)$
⟨*proof*⟩

corollary *bellman_ford_shortest_paths*:
assumes ¬ *has_negative_cycle*
shows $\forall v \leq n.$ *bf* n $v =$ *shortest* v
⟨*proof*⟩

lemma *OPT_mono*:
OPT m $v \leq$ *OPT* n v **if** ‹$v \leq n$› ‹$n \leq m$›
⟨*proof*⟩

corollary *bf_fix*:
assumes ¬ *has_negative_cycle* $m \geq n$
shows $\forall v \leq n.$ *bf* m $v =$ *bf* n v
⟨*proof*⟩

lemma *bellman_ford_correct′*:
bf_m.*crel_vs* (=) (*if has_negative_cycle then None else Some* (*map shortest* [0..<n+1])) *bellman_ford*

⟨*proof*⟩
include *state_monad_syntax app_syntax*
⟨*proof*⟩

theorem *bellman_ford_correct*:
fst (*run_state bellman_ford Mapping.empty*) =
(*if has_negative_cycle then None else Some* (*map shortest* [0..<*n*+1]))
⟨*proof*⟩

end

end

end

3.2.7 Extracting an Executable Constant for the Imperative Implementation

ground_function (*prove_termination*) bf_h'*_impl*: bf_h'*.simps*

lemma bf_h'*_impl_def*:
fixes *n* :: *nat*
fixes *mem* :: *nat ref* × *nat ref* × *int extended option array ref* × *int extended option array ref*
assumes *mem_is_init*: *mem* = *result_of* (*init_state* (*n* + 1) 1 0) *Heap.empty*
shows bf_h'*_impl n w t mem* = bf_h' *n w t mem*
⟨*proof*⟩

definition
iter_bf_heap n w t mem = *iterator_defs.iter_heap*
(λ(*x*, *y*). *x* ≤ *n* ∧ *y* ≤ *n*)
(λ(*x*, *y*). *if y* < *n then* (*x*, *y* + 1) *else* (*x* + 1, 0))
(λ(*x*, *y*). bf_h'*_impl n w t mem x y*)

lemma *iter_bf_heap_unfold*[*code*]:
iter_bf_heap n w t mem = (λ (*i*, *j*).
(*if i* ≤ *n* ∧ *j* ≤ *n*
then do {
bf_h'*_impl n w t mem i j*;
iter_bf_heap n w t mem (*if j* < *n then* (*i*, *j* + 1) *else* (*i* + 1, 0))
}
else Heap_Monad.return ()))
⟨*proof*⟩

definition
bf_impl n w t i j = do {
mem ← (init_state (n + 1) (1::nat) (0::nat) ::
(nat ref × nat ref × int extended option array ref × int extended option array ref) Heap);
iter_bf_heap n w t mem (0, 0);
bf$_h$'_impl n w t mem i j
}

lemma *bf_impl_correct*:
bf n w t i j = result_of (bf_impl n w t i j) Heap.empty
⟨*proof*⟩

3.2.8 Test Cases

definition
G$_1$_list = [[(1 :: *nat*,−6 :: *int*), (2,4), (3,5)], [(3,10)], [(3,2)], []]

definition
G$_2$_list = [[(1 :: *nat*,−6 :: *int*), (2,4), (3,5)], [(3,10)], [(3,2)], [(0, −5)]]

definition
G$_3$_list = [[(1 :: *nat*,−1 :: *int*), (2,2)], [(2,5), (3,4)], [(3,2), (4,3)], [(2,−2), (4,2)], []]

definition
G$_4$_list = [[(1 :: *nat*,−1 :: *int*), (2,2)], [(2,5), (3,4)], [(3,2), (4,3)], [(2,−3), (4,2)], []]

definition
graph_of a i j = case_option ∞ (Fin o snd) (List.find (λ p. fst p = j) (a !! i))

definition *test_bf = bf_impl* 3 (*graph_of* (*IArray G$_1$_list*)) 3 3 0

code_reflect *Test* **functions** *test_bf*

One can see a trace of the calls to the memory in the output

⟨*ML*⟩

lemma *bottom_up_alt*[*code*]:
bf n W t i j =
fst (run_state
(iter_bf n W t (0, 0) ≫= (λ_. bf$_m$' n W t i j))

Mapping.empty)
⟨*proof*⟩

definition
bf_ia n W t i j = (*let W'* = *graph_of* (*IArray W*) *in*
fst (*run_state*
(*iter_bf n W' t* (*i*, *j*) ⤜ (λ_. bf_m' *n W' t i j*))
Mapping.empty)
)

— Component tests.
lemma
fst (*run_state* (bf_m' 3 (*graph_of* (*IArray* G_1*_list*)) 3 3 0) *Mapping.empty*)
= 4
bf 3 (*graph_of* (*IArray* G_1*_list*)) 3 3 0 = 4
⟨*proof*⟩
lemma
fst (*run_state* (*bellman_ford* 3 (*graph_of* (*IArray* G_1*_list*)) 3) *Mapping.empty*) = *Some* [4, 10, 2, 0]
fst (*run_state* (*bellman_ford* 4 (*graph_of* (*IArray* G_3*_list*)) 4) *Mapping.empty*) = *Some* [4, 5, 3, 1, 0]
⟨*proof*⟩
lemma
fst (*run_state* (*bellman_ford* 3 (*graph_of* (*IArray* G_2*_list*)) 3) *Mapping.empty*) = *None*
fst (*run_state* (*bellman_ford* 4 (*graph_of* (*IArray* G_4*_list*)) 4) *Mapping.empty*) = *None*
⟨*proof*⟩

end
theory *Heap_Default*
imports
Heap_Main
../Indexing
begin

locale *dp_consistency_heap_default* =
fixes *bound* :: *'k* :: {*index*, *heap*} *bound*
and *mem* :: *'v*::*heap option array*
and *dp* :: *'k* ⇒ *'v*
begin

interpretation *idx*: *bounded_index bound* ⟨*proof*⟩

sublocale *dp_consistency_heap*
  where *P=λheap. Array.length heap mem = idx.size*
    and *lookup=mem_lookup idx.size idx.checked_idx mem*
    and *update=mem_update idx.size idx.checked_idx mem*
  ⟨*proof*⟩

context
  fixes *empty*
  assumes *empty*: *map_of_heap empty* $\subseteq_m$ *Map.empty*
      and *len*: *Array.length empty mem = idx.size*
begin

interpretation *consistent*: *dp_consistency_heap_empty*
  where *P=λheap. Array.length heap mem = idx.size*
    and *lookup=mem_lookup idx.size idx.checked_idx mem*
    and *update=mem_update idx.size idx.checked_idx mem*
  ⟨*proof*⟩

lemmas *memoizedI = consistent.memoized*
lemmas *successI = consistent.memoized_success*

end

lemma *mem_empty_empty*:
  map_of_heap (*heap_of* (*mem_empty idx.size* :: *'v option array Heap*) *Heap.empty*) $\subseteq_m$ *Map.empty*
  if *mem = result_of* (*mem_empty idx.size*) *Heap.empty*
  ⟨*proof*⟩

lemma *memoized_empty*:
  dp x = result_of ((*mem_empty idx.size* :: *'v option array Heap*) $\gg=$ (*λmem. dp*$_T$ *mem x*)) *Heap.empty*
  if *consistentDP* (*dp*$_T$ *mem*) *mem = result_of* (*mem_empty idx.size*) *Heap.empty*
  ⟨*proof*⟩

lemma *init_success*:
  success ((*mem_empty idx.size* :: *'v option array Heap*) $\gg=$ (*λmem. dp*$_T$ *mem x*)) *Heap.empty*
  if *consistentDP* (*dp*$_T$ *mem*) *mem = result_of* (*mem_empty idx.size*) *Heap.empty*
  ⟨*proof*⟩

end

end

3.3 The Knapsack Problem

theory *Knapsack*
imports
HOL−Library.Code_Target_Numeral
../state_monad/State_Main
../heap_monad/Heap_Default
Example_Misc
begin

3.3.1 Definitions

context
fixes $w :: nat \Rightarrow nat$
begin

context
fixes $v :: nat \Rightarrow nat$
begin

fun *knapsack* :: $nat \Rightarrow nat \Rightarrow nat$ **where**
knapsack 0 $W = 0$ |
knapsack (*Suc i*) W = (*if* $W < w$ (*Suc i*)
then knapsack i W
else max (*knapsack i W*) (v (*Suc i*) + *knapsack i* ($W - w$ (*Suc i*))))

no_notation *fun_app_lifted* (**infixl** . 999)

The correctness proof closely follows Kleinberg & Tardos: "Algorithm Design", chapter "Dynamic Programming" [1]

definition
$OPT\ n\ W = Max\ \{\sum\ i \in S.\ v\ i \mid S.\ S \subseteq \{1..n\} \wedge (\sum\ i \in S.\ w\ i) \leq W\}$

lemma *OPT_0*:
$OPT\ 0\ W = 0$
⟨*proof*⟩

3.3.2 Functional Correctness

lemma *Max_add_left*:
$(x :: nat) + Max\ S = Max\ (((+)\ x)\ `\ S)$ (**is** *?A* = *?B*) **if** *finite S* $S \neq \{\}$

⟨*proof*⟩

lemma *OPT_Suc*:
  OPT (*Suc i*) *W* = (
    if W < *w* (*Suc i*)
    then OPT i W
    else max(*v* (*Suc i*) + *OPT i* (*W* − *w* (*Suc i*))) (*OPT i W*)
) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

theorem *knapsack_correct*:
  OPT n W = *knapsack n W*
  ⟨*proof*⟩

3.3.3 Functional Memoization

memoize_fun *knapsack*$_m$: *knapsack* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *knapsack.simps*

Generated Definitions

context includes *state_monad_syntax* **begin**
thm *knapsack*$_m$*′.simps knapsack*$_m$*_def*
end

Correspondence Proof

memoize_correct
  ⟨*proof*⟩
print_theorems
lemmas [*code*] = *knapsack*$_m$*.memoized_correct*

3.3.4 Imperative Memoization

context fixes
  mem :: *nat option array*
  and *n W* :: *nat*
begin

memoize_fun *knapsack*$_T$: *knapsack*
  with_memory *dp_consistency_heap_default* **where** *bound* = *Bound*
(0, 0) (*n*, *W*) **and** *mem*=*mem*
  monadifies (*heap*) *knapsack.simps*

context includes *heap_monad_syntax* **begin**
thm *knapsack*$_T$*′.simps knapsack*$_T$*_def*

end

memoize_correct
⟨*proof*⟩

lemmas *memoized_empty* = $knapsack_T$.*memoized_empty*

end

Adding Memory Initialization

context
includes *heap_monad_syntax*
notes [*simp del*] = $knapsack_T{}'$.*simps*
begin

definition
$knapsack_h \equiv \lambda\ i\ j.$ *Heap_Monad.bind* (*mem_empty* (*i* ∗ *j*)) (λ *mem.* $knapsack_T{}'$ *mem i j i j*)

lemmas *memoized_empty*′ = *memoized_empty*[
of mem n W λ *m.* λ(*i*,*j*). $knapsack_T{}'$ *m n W i j*,
OF $knapsack_T$.*crel*[*of mem n W*], *of* (*n*, *W*) **for** *mem n W*
]

lemma *knapsack_heap*:
knapsack n W = *result_of* ($knapsack_h$ *n W*) *Heap.empty*
⟨*proof*⟩

end

end

fun *su* :: *nat* ⇒ *nat* ⇒ *nat* **where**
su 0 *W* = 0 |
su (*Suc i*) *W* = (*if W* < *w* (*Suc i*)
then su i W
else max (*su i W*) (*w* (*Suc i*) + *su i* (*W* − *w* (*Suc i*))))

lemma *su_knapsack*:
su n W = *knapsack w n W*
⟨*proof*⟩

lemma *su_correct*:
Max {∑ *i* ∈ *S*. *w i* | *S*. *S* ⊆ {1..*n*} ∧ (∑ *i* ∈ *S*. *w i*) ≤ *W*} = *su n W*

⟨*proof*⟩

3.3.5 Memoization

memoize_fun su_m: *su* **with_memory** *dp_consistency_mapping* **monadifies** (*state*) *su.simps*

Generated Definitions

context includes *state_monad_syntax* **begin**
thm su_m'*.simps* su_m*_def*
end

Correspondence Proof

memoize_correct
⟨*proof*⟩
print_theorems
lemmas [*code*] = su_m*.memoized_correct*

end

3.3.6 Regression Test

definition
knapsack_test = (*knapsack*$_h$ (λ *i*. [2,3,4] ! (*i* − 1)) (λ *i*. [2,3,4] ! (*i* − 1)) 3 8)

code_reflect *Test* **functions** *knapsack_test*

⟨*ML*⟩

end
theory *Counting_Tiles*
imports
HOL−Library.Code_Target_Numeral
HOL−Library.Product_Lexorder
HOL−Library.RBT_Mapping
../state_monad/State_Main
Example_Misc
begin

3.4 A Counting Problem

This formalization contains verified solutions for Project Euler problems

- #114 (`https://projecteuler.net/problem=114`) and

- #115 (`https://projecteuler.net/problem=115`).

This is the problem description for #115:

> A row measuring n units in length has red blocks with a minimum length of m units placed on it, such that any two red blocks (which are allowed to be different lengths) are separated by at least one black square. Let the fill-count function, F(m, n), represent the number of ways that a row can be filled.
>
> For example, F(3, 29) = 673135 and F(3, 30) = 1089155.
>
> That is, for m = 3, it can be seen that n = 30 is the smallest value for which the fill-count function first exceeds one million. In the same way, for m = 10, it can be verified that F(10, 56) = 880711 and F(10, 57) = 1148904, so n = 57 is the least value for which the fill-count function first exceeds one million.
>
> For m = 50, find the least value of n for which the fill-count function first exceeds one million.

3.4.1 Misc

lemma *lists_of_len_fin*1:
finite (*lists A* ∩ {*l*. *length l* = *n*}) **if** *finite A*
⟨*proof*⟩

lemma *disjE*1:
A ∨ *B* ⟹ (*A* ⟹ *P*) ⟹ (¬ *A* ⟹ *B* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

3.4.2 Problem Specification

Colors

datatype *color* = *R* | *B*

Direct natural definition of a valid line

context
fixes *m* :: *nat*
begin

inductive *valid* **where**
valid [] |
valid xs ⟹ *valid* (*B* # *xs*) |

valid xs $\Longrightarrow$ $n \geq m$ $\Longrightarrow$ *valid* (*replicate n R* @ *xs*)

Definition of the fill-count function

definition $F\ n$ = *card* $\{l.\ length\ l = n \wedge valid\ l\}$

3.4.3 Combinatorial Identities

This alternative variant helps us to prove the split lemma below.

inductive *valid′* **where**
valid′ [] |
$n \geq m \Longrightarrow$ *valid′* (*replicate n R*) |
valid′ xs $\Longrightarrow$ *valid′* (*B* # *xs*) |
valid′ xs $\Longrightarrow$ $n \geq m \Longrightarrow$ *valid′* (*replicate n R* @ *B* # *xs*)

lemma *valid_valid′*:
valid l $\Longrightarrow$ *valid′ l*
⟨*proof*⟩

lemmas *valid_red* = *valid.intros*(3)[*OF valid.intros*(1), *simplified*]

lemma *valid′_valid*:
valid′ l $\Longrightarrow$ *valid l*
⟨*proof*⟩

lemma *valid_eq_valid′*:
valid′ l = *valid l*
⟨*proof*⟩

Additional Facts on Replicate

lemma *replicate_iff*:
(∀ *i*<*length l*. *l* ! *i* = *R*) $\longleftrightarrow$ (∃ *n*. *l* = *replicate n R*)
⟨*proof*⟩

lemma *replicate_iff*2:
(∀ *i*<*n*. *l* ! *i* = *R*) $\longleftrightarrow$ (∃ *l′*. *l* = *replicate n R* @ *l′*) **if** $n < length\ l$
⟨*proof*⟩

lemma *replicate_Cons_eq*:
replicate n x = *y* # *ys* $\longleftrightarrow$ (∃ *n′*. *n* = *Suc n′* ∧ *x* = *y* ∧ *replicate n′ x* = *ys*)
⟨*proof*⟩

Main Case Analysis on @*term valid*

lemma *valid_split*:

valid l $\longleftrightarrow$
l = [] $\vee$
(*l*!0 = *B* $\wedge$ *valid* (*tl l*)) $\vee$
length l $\geq$ *m* $\wedge$ ($\forall$ *i* < *length l*. *l* ! *i* = *R*) $\vee$
($\exists$ *j* < *length l*. *j* $\geq$ *m* $\wedge$ ($\forall$ *i* < *j*. *l* ! *i* = *R*) $\wedge$ *l* ! *j* = *B* $\wedge$ *valid* (*drop* (*j* + 1) *l*))
⟨*proof*⟩

Base cases

lemma *valid_line_just_B*:
valid (*replicate n B*)
⟨*proof*⟩

lemma *F_base_0_aux*:
{*l*. *l* = [] $\wedge$ *valid l*} = {[]}
⟨*proof*⟩

lemma *F_base_0*: *F* 0 = 1
⟨*proof*⟩

lemma *F_base_aux*: {*l*. *length l*=*n* $\wedge$ *valid l*} = {*replicate n B*} **if** *n* > 0 *n* < *m*
⟨*proof*⟩

lemma *F_base_1*:
F n = 1 **if** *n* > 0 *n* < *m*
⟨*proof*⟩

lemma *valid_m_Rs* [*simp*]:
valid (*replicate m R*)
⟨*proof*⟩

lemma *F_base_aux_2*: {*l*. *length l*=*m* $\wedge$ *valid l*} = {*replicate m R*, *replicate m B*}
⟨*proof*⟩

lemma *F_base_2*:
F m = 2 **if** 0 < *m*
⟨*proof*⟩

The recursion case

lemma *finite_valid_length*:
finite {*l*. *length l* = *n* $\wedge$ *valid l*} (**is** *finite ?S*)
⟨*proof*⟩

lemma *valid_line_aux*:
  $\{l.\ length\ l = n \wedge valid\ l\} \neq \{\}$ (**is** *?S* $\neq \{\}$)
  ⟨*proof*⟩

lemma *replicate_unequal_aux*:
  replicate x R @ B # l $\neq$ *replicate y R @ B # l'* (**is** *?l* $\neq$ *?r*) **if** ‹$x < y$› **for** *l l'*
⟨*proof*⟩

lemma *valid_prepend_B_iff*:
  valid (*B # xs*) $\longleftrightarrow$ *valid xs* **if** $m > 0$
  ⟨*proof*⟩

lemma *F_rec*: *F n* = *F* (*n*−1) + 1 + ($\sum$ *i*=*m*..<*n*. *F* (*n*−*i*−1)) **if** ‹*n*>*m*› $m > 0$
⟨*proof*⟩

3.4.4 Computing the Fill-Count Function

fun *lcount* :: *nat* ⇒ *nat* **where**
  lcount n = (
    if n < *m then* 1
    else if n = *m then* 2
    else lcount (*n* − 1) + 1 + ($\sum$ *i* ← [*m*..<*n*]. *lcount* (*n* − *i* − 1))
)

lemmas [*simp del*] = *lcount.simps*

lemma *lcount_correct*:
  lcount n = *F n* **if** $m > 0$
⟨*proof*⟩

3.4.5 Memoization

memoize_fun *lcount*$_m$: *lcount* **with_memory** *dp_consistency_mapping* **monadifies** (*state*) *lcount.simps*

memoize_correct
  ⟨*proof*⟩

lemmas [*code*] = *lcount*$_m$*.memoized_correct*

end

3.4.6 Problem solutions

Example and solution for problem #114

value *lcount* 3 7
value *lcount* 3 50

Examples for problem #115

value *lcount* 3 29
value *lcount* 3 30
value *lcount* 10 56
value *lcount* 10 57

Binary search for the solution of problem #115

value *lcount* 50 100
value *lcount* 50 150
value *lcount* 50 163
value *lcount* 50 166
value *lcount* 50 167
value *lcount* 50 168 — The solution
value *lcount* 50 169
value *lcount* 50 175
value *lcount* 50 200
value *lcount* 50 300
value *lcount* 50 500
value *lcount* 50 1000

We prove that 168 is the solution for problem #115

theorem
(*LEAST n*. *F* 50 *n* > 1000000) = 168
⟨*proof*⟩

end

3.5 The CYK Algorithm

theory *CYK*
imports
HOL−Library.IArray
HOL−Library.Code_Target_Numeral
HOL−Library.Product_Lexorder
HOL−Library.RBT_Mapping
../state_monad/State_Main
../heap_monad/Heap_Default
Example_Misc
begin

3.5.1 Misc

lemma *append_iff_take_drop*:
$w = u@v \longleftrightarrow (\exists\, k \in \{0..length\ w\}.\ u = take\ k\ w \wedge v = drop\ k\ w)$
⟨*proof*⟩

lemma *append_iff_take_drop*1: $u \neq [] \Longrightarrow v \neq [] \Longrightarrow$
$w = u@v \longleftrightarrow (\exists\, k \in \{1..length\ w - 1\}.\ u = take\ k\ w \wedge v = drop\ k\ w)$
⟨*proof*⟩

3.5.2 Definitions

datatype $('n, 't)\ rhs = NN\ 'n\ 'n \mid T\ 't$

type_synonym $('n, 't)\ prods = ('n \times ('n, 't)\ rhs)\ list$

context
fixes $P :: ('n :: heap, 't)\ prods$
begin

inductive $yield :: 'n \Rightarrow 't\ list \Rightarrow bool$ **where**
$(A, T\ a) \in set\ P \Longrightarrow yield\ A\ [a] \mid$
$[\![\ (A, NN\ B\ C) \in set\ P; yield\ B\ u; yield\ C\ v\ ]\!] \Longrightarrow yield\ A\ (u@v)$

lemma *yield_not_Nil*: $yield\ A\ w \Longrightarrow w \neq []$
⟨*proof*⟩

lemma *yield_eq*1:
$yield\ A\ [a] \longleftrightarrow (A, T\ a) \in set\ P$ (**is** $?L = ?R$)
⟨*proof*⟩

lemma *yield_eq*2: **assumes** $length\ w > 1$
shows $yield\ A\ w \longleftrightarrow (\exists\, B\ u\ C\ v.\ yield\ B\ u \wedge yield\ C\ v \wedge w = u@v \wedge (A, NN\ B\ C) \in set\ P)$
(**is** $?L = ?R$)
⟨*proof*⟩

3.5.3 CYK on Lists

fun $cyk :: 't\ list \Rightarrow 'n\ list$ **where**
$cyk\ [] = [] \mid$
$cyk\ [a] = [A\ .\ (A, T\ a') <- P, a' = a] \mid$
$cyk\ w =$
$[A.\ k <- [1..<length\ w], B <- cyk\ (take\ k\ w), C <- cyk\ (drop\ k\ w), (A, NN\ B'\ C') <- P, B' = B, C' = C]$

lemma *set_cyk_simp2*[*simp*]: *length w* ≥ 2 ⟹ *set*(*cyk w*) =
(⋃ *k* ∈ {1..*length w* − 1}. ⋃ *B* ∈ *set*(*cyk* (*take k w*)). ⋃ *C* ∈ *set*(*cyk* (*drop k w*)). {*A*. (*A*, *NN B C*) ∈ *set P*})
⟨*proof*⟩

declare *cyk.simps*(3)[*simp del*]

lemma *cyk_correct*: *set*(*cyk w*) = {*N*. *yield N w*}
⟨*proof*⟩

3.5.4 CYK on Lists and Index

fun *cyk2* :: *'t list* ⇒ *nat* ∗ *nat* ⇒ *'n list* **where**
cyk2 w (*i*,0) = [] |
cyk2 w (*i*,*Suc* 0) = [*A* . (*A*, *T a*) <− *P*, *a* = *w*!*i*] |
cyk2 w (*i*,*n*) =
[*A*. *k* <− [1..<*n*], *B* <− *cyk2 w* (*i*,*k*), *C* <− *cyk2 w* (*i*+*k*,*n*−*k*), (*A*, *NN B' C'*) <− *P*, *B'* = *B*, *C'* = *C*]

lemma *set_aux*: (⋃ *xb*∈*set P*. {*A*. (*A*, *NN B C*) = *xb*}) = {*A*. (*A*, *NN B C*) ∈ *set P*}
⟨*proof*⟩

lemma *cyk2_eq_cyk*: *i*+*n* ≤ *length w* ⟹ *set*(*cyk2 w* (*i*,*n*)) = *set*(*cyk* (*take n* (*drop i w*)))
⟨*proof*⟩

definition *CYK S w* = (*S* ∈ *set*(*cyk2 w* (0, *length w*)))

theorem *CYK_correct*: *CYK S w* = *yield S w*
⟨*proof*⟩

3.5.5 CYK With Index Function

context
fixes *w* :: *nat* ⇒ *'t*
begin

fun *cyk_ix* :: *nat* ∗ *nat* ⇒ *'n list* **where**
cyk_ix (*i*,0) = [] |
cyk_ix (*i*,*Suc* 0) = [*A* . (*A*, *T a*) <− *P*, *a* = *w i*] |
cyk_ix (*i*,*n*) =
[*A*. *k* <− [1..<*n*], *B* <− *cyk_ix* (*i*,*k*), *C* <− *cyk_ix* (*i*+*k*,*n*−*k*), (*A*, *NN*

$B'\ C'$) <− P, $B' = B$, $C' = C$]

3.5.6 Correctness Proof

lemma *cyk_ix_simp*2: *set*(*cyk_ix* (*i*,*Suc*(*Suc n*))) =
(⋃ $k \in \{1..Suc\ n\}$. ⋃ $B \in set(cyk\_ix\ (i,k))$. ⋃ $C \in set(cyk\_ix\ (i{+}k,n{+}2{-}k))$.
$\{A.\ (A,\ NN\ B\ C) \in set\ P\}$)
⟨*proof*⟩

declare *cyk_ix.simps*(3)[*simp del*]

abbreviation (*input*) *slice f i j* ≡ *map f* [*i*..<*j*]

lemma *slice_append_iff_take_drop*1: $u \neq [] \Longrightarrow v \neq [] \Longrightarrow$
slice w i j = *u* @ *v* ⟷ (∃ *k*. 1 ≤ *k* ∧ *k* ≤ *j*−*i*−1 ∧ *slice w i* (*i* + *k*) = *u*
∧ *slice w* (*i* + *k*) *j* = *v*)
⟨*proof*⟩

lemma *cyk_ix_correct*:
set(*cyk_ix* (*i*,*n*)) = {*N*. *yield N* (*slice w i* (*i*+*n*))}
⟨*proof*⟩

3.5.7 Functional Memoization

memoize_fun *cyk_ix*$_m$: *cyk_ix* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *cyk_ix.simps*
thm *cyk_ix*$_m$′*.simps*

memoize_correct
⟨*proof*⟩
print_theorems

lemmas [*code*] = *cyk_ix*$_m$*.memoized_correct*

3.5.8 Imperative Memoization

context
fixes *n* :: *nat*
begin

context
fixes *mem* :: ′*n list option array*
begin

memoize_fun *cyk_ix*$_h$: *cyk_ix*

with_memory *dp_consistency_heap_default* **where** *bound* = *Bound* (0, 0) (*n*, *n*) **and** *mem*=*mem*
monadifies (*heap*) *cyk_ix.simps*

context includes *heap_monad_syntax* **begin**
thm *cyk_ix*$_h$′*.simps cyk_ix*$_h$*_def*
end

memoize_correct
⟨*proof*⟩

lemmas *memoized_empty* = *cyk_ix*$_h$*.memoized_empty*

lemmas *init_success* = *cyk_ix*$_h$*.init_success*

end

definition *cyk_ix_impl i j* = *do* {*mem* ← *mem_empty* (*n* ∗ *n*); *cyk_ix*$_h$′ *mem* (*i*, *j*)}

lemma *cyk_ix_impl_success*:
success (*cyk_ix_impl i j*) *Heap.empty*
⟨*proof*⟩

lemma *min_wpl_heap*:
cyk_ix (*i*, *j*) = *result_of* (*cyk_ix_impl i j*) *Heap.empty*
⟨*proof*⟩

end

end

definition *CYK_ix S w n* = (*S* ∈ *set*(*cyk_ix w* (0,*n*)))

theorem *CYK_ix_correct*: *CYK_ix S w n* = *yield S* (*slice w* 0 *n*)
⟨*proof*⟩

definition *cyk_list w* = *cyk_ix* (λ*i*. *w* ! *i*) (0,*length w*)

definition
CYK_ix_impl S w n = *do* {*R* ← *cyk_ix_impl w n* 0 *n*; *return* (*S* ∈ *set R*)}

lemma *CYK_ix_impl_correct*:

result_of (*CYK_ix_impl S w n*) *Heap.empty* = *yield S* (*slice w* 0 *n*)
⟨*proof*⟩

end

3.5.9 Functional Test Case

value
(*let P* = [(0::*int*, *NN* 1 2), (0, *NN* 2 3),
(1, *NN* 2 1), (1, *T* (*CHR* ″*a*″)),
(2, *NN* 3 3), (2, *T* (*CHR* ″*b*″)),
(3, *NN* 1 2), (3, *T* (*CHR* ″*a*″))]
in map (λ*w*. *cyk*2 *P w* (0,*length w*)) [″*baaba*″, ″*baba*″])

value
(*let P* = [(0::*int*, *NN* 1 2), (0, *NN* 2 3),
(1, *NN* 2 1), (1, *T* (*CHR* ″*a*″)),
(2, *NN* 3 3), (2, *T* (*CHR* ″*b*″)),
(3, *NN* 1 2), (3, *T* (*CHR* ″*a*″))]
in map (*cyk_list P*) [″*baaba*″, ″*baba*″])

definition *cyk_ia P w* = (*let a* = *IArray w in cyk_ix P* (λ*i*. *a* !! *i*) (0,*length w*))

value
(*let P* = [(0::*int*, *NN* 1 2), (0, *NN* 2 3),
(1, *NN* 2 1), (1, *T* (*CHR* ″*a*″)),
(2, *NN* 3 3), (2, *T* (*CHR* ″*b*″)),
(3, *NN* 1 2), (3, *T* (*CHR* ″*a*″))]
in map (*cyk_ia P*) [″*baaba*″, ″*baba*″])

3.5.10 Imperative Test Case

definition *cyk_ia′ P w* = (*let a* = *IArray w in cyk_ix_impl P* (λ*i*. *a* !! *i*) (*length w*) 0 (*length w*))

definition
test = (*let P* = [(0::*int*, *NN* 1 2), (0, *NN* 2 3),
(1, *NN* 2 1), (1, *T* (*CHR* ″*a*″)),
(2, *NN* 3 3), (2, *T* (*CHR* ″*b*″)),
(3, *NN* 1 2), (3, *T* (*CHR* ″*a*″))]
in map (*cyk_ia′ P*) [″*baaba*″, ″*baba*″])

code_reflect *Test* **functions** *test*

⟨*ML*⟩

end

3.6 Minimum Edit Distance

theory *Min_Ed_Dist*0
imports
HOL−Library.IArray
HOL−Library.Code_Target_Numeral
HOL−Library.Product_Lexorder
HOL−Library.RBT_Mapping
../state_monad/State_Main
../heap_monad/Heap_Main
Example_Misc
../util/Tracing
../util/Ground_Function
begin

3.6.1 Misc

Executable argmin

fun *argmin* :: (′*a* ⇒ ′*b*::*order*) ⇒ ′*a list* ⇒ ′*a* **where**
argmin f [*a*] = *a* |
argmin f (*a*#*as*) = (*let m* = *argmin f as in if f a* ≤ *f m then a else m*)

fun *argmin*2 :: (′*a* ⇒ ′*b*::*order*) ⇒ ′*a list* ⇒ ′*a* ∗ ′*b* **where**
*argmin*2 *f* [*a*] = (*a*, *f a*) |
*argmin*2 *f* (*a*#*as*) = (*let fa* = *f a*; (*am*,*m*) = *argmin*2 *f as in if fa* ≤ *m then* (*a*, *fa*) *else* (*am*,*m*))

3.6.2 Edit Distance

datatype ′*a ed* = *Copy* | *Repl* ′*a* | *Ins* ′*a* | *Del*

fun *edit* :: ′*a ed list* ⇒ ′*a list* ⇒ ′*a list* **where**
edit (*Copy* # *es*) (*x* # *xs*) = *x* # *edit es xs* |
edit (*Repl a* # *es*) (*x* # *xs*) = *a* # *edit es xs* |
edit (*Ins a* # *es*) *xs* = *a* # *edit es xs* |
edit (*Del* # *es*) (*x* # *xs*) = *edit es xs* |
edit (*Copy* # *es*) [] = *edit es* [] |

edit (*Repl a* # *es*) [] = *edit es* [] |
edit (*Del* # *es*) [] = *edit es* [] |
edit [] *xs* = *xs*

abbreviation *cost* **where**
cost es ≡ *length* [*e* <− *es*. *e* ≠ *Copy*]

3.6.3 Minimum Edit Sequence

fun *min_eds* :: *'a list* ⇒ *'a list* ⇒ *'a ed list* **where**
min_eds [] [] = [] |
min_eds [] (*y*#*ys*) = *Ins y* # *min_eds* [] *ys* |
min_eds (*x*#*xs*) [] = *Del* # *min_eds xs* [] |
min_eds (*x*#*xs*) (*y*#*ys*) =
  argmin cost [*Ins y* # *min_eds* (*x*#*xs*) *ys*, *Del* # *min_eds xs* (*y*#*ys*),
    (*if x*=*y then Copy else Repl y*) # *min_eds xs ys*]

lemma *min_eds ''vintner'' ''writers'' =*
  [*Ins CHR ''w''*, *Repl CHR ''r''*, *Copy*, *Del*, *Copy*, *Del*, *Copy*, *Copy*, *Ins CHR ''s''*]
⟨*proof*⟩

lemma *min_eds_correct*: *edit* (*min_eds xs ys*) *xs* = *ys*
⟨*proof*⟩

lemma *min_eds_same*: *min_eds xs xs* = *replicate* (*length xs*) *Copy*
⟨*proof*⟩

lemma *min_eds_eq_Nil_iff*: *min_eds xs ys* = [] ⟷ *xs* = [] ∧ *ys* = []
⟨*proof*⟩

lemma *min_eds_Nil*: *min_eds* [] *ys* = *map Ins ys*
⟨*proof*⟩

lemma *min_eds_Nil2*: *min_eds xs* [] = *replicate* (*length xs*) *Del*
⟨*proof*⟩

lemma *if_edit_Nil2*: *edit es* ([]::*'a list*) = *ys* ⟹ *length ys* ≤ *cost es*
⟨*proof*⟩

lemma *if_edit_eq_Nil*: *edit es xs* = [] ⟹ *length xs* ≤ *cost es*
⟨*proof*⟩

lemma *min_eds_minimal*: *edit es xs* = *ys* ⟹ *cost*(*min_eds xs ys*) ≤ *cost*

es
⟨*proof*⟩

3.6.4 Computing the Minimum Edit Distance

fun *min_ed* :: *'a list* ⇒ *'a list* ⇒ *nat* **where**
min_ed [] [] = 0 |
min_ed [] (*y*#*ys*) = 1 + *min_ed* [] *ys* |
min_ed (*x*#*xs*) [] = 1 + *min_ed xs* [] |
min_ed (*x*#*xs*) (*y*#*ys*) =
Min {1 + *min_ed* (*x*#*xs*) *ys*, 1 + *min_ed xs* (*y*#*ys*), (*if x*=*y then* 0 *else* 1) + *min_ed xs ys*}

lemma *min_ed_min_eds*: *min_ed xs ys* = *cost*(*min_eds xs ys*)
⟨*proof*⟩

lemma *min_ed ''madagascar'' ''bananas''* = 6
⟨*proof*⟩

Exercise: Optimization of the Copy case

fun *min_eds2* :: *'a list* ⇒ *'a list* ⇒ *'a ed list* **where**
min_eds2 [] [] = [] |
min_eds2 [] (*y*#*ys*) = *Ins y* # *min_eds2* [] *ys* |
min_eds2 (*x*#*xs*) [] = *Del* # *min_eds2 xs* [] |
min_eds2 (*x*#*xs*) (*y*#*ys*) =
(*if x*=*y then Copy* # *min_eds2 xs ys*
else argmin cost
[*Ins y* # *min_eds2* (*x*#*xs*) *ys*, *Del* # *min_eds2 xs* (*y*#*ys*), *Repl y* # *min_eds2 xs ys*])

value *min_eds2 ''madagascar'' ''bananas''*

lemma *cost_Copy_Del*: *cost*(*min_eds xs ys*) ≤ *cost* (*min_eds xs* (*x*#*ys*)) + 1
⟨*proof*⟩

lemma *cost_Copy_Ins*: *cost*(*min_eds xs ys*) ≤ *cost* (*min_eds* (*x*#*xs*) *ys*) + 1
⟨*proof*⟩

lemma *cost*(*min_eds2 xs ys*) = *cost*(*min_eds xs ys*)
⟨*proof*⟩

lemma *min_eds2 xs ys* = *min_eds xs ys*

⟨*proof*⟩

3.6.5 Indexing

Indexing lists

context
fixes *xs ys* :: *'a list*
fixes *m n* :: *nat*
begin

function (*sequential*)
  min_ed_ix' :: *nat* ∗ *nat* ⇒ *nat* **where**
min_ed_ix' (i,j) =
  (*if i* ≥ *m then*
     if j ≥ *n then* 0 *else* 1 + *min_ed_ix' (i,j*+1) *else*
   if j ≥ *n then* 1 + *min_ed_ix' (i*+1, *j*)
   else
   Min {1 + *min_ed_ix' (i,j*+1), 1 + *min_ed_ix' (i*+1, *j*),
      (*if xs*!*i* = *ys*!*j then* 0 *else* 1) + *min_ed_ix' (i*+1,*j*+1)})
⟨*proof*⟩
termination ⟨*proof*⟩

declare *min_ed_ix'.simps*[*simp del*]

end

lemma *min_ed_ix'_min_ed*:
  min_ed_ix' xs ys (*length xs*) (*length ys*) (*i*, *j*) = *min_ed* (*drop i xs*) (*drop j ys*)
⟨*proof*⟩

Indexing functions

context
fixes *xs ys* :: *nat* ⇒ *'a*
fixes *m n* :: *nat*
begin

function (*sequential*)
  min_ed_ix :: *nat* × *nat* ⇒ *nat* **where**
min_ed_ix (*i*, *j*) =
  (*if i* ≥ *m then*
     if j ≥ *n then* 0 *else n*−*j else*
   if j ≥ *n then m*−*i*

else
min_list [1 + *min_ed_ix* (*i*, *j*+1), 1 + *min_ed_ix* (*i*+1, *j*),
(*if xs i* = *ys j then* 0 *else* 1) + *min_ed_ix* (*i*+1, *j*+1)])
⟨*proof*⟩
termination ⟨*proof*⟩

3.6.6 Functional Memoization

memoize_fun *min_ed_ix*$_m$: *min_ed_ix* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *min_ed_ix.simps*
thm *min_ed_ix*$_m$′*.simps*

memoize_correct
⟨*proof*⟩
print_theorems

lemmas [*code*] = *min_ed_ix*$_m$*.memoized_correct*

declare *min_ed_ix.simps*[*simp del*]

3.6.7 Imperative Memoization

context
fixes *mem* :: *nat ref* × *nat ref* × *nat option array ref* × *nat option array ref*
assumes *mem_is_init*: *mem* = *result_of* (*init_state* (*n* + 1) *m* (*m* + 1)) *Heap.empty*
begin

interpretation *iterator*
λ (*x*, *y*). *x* ≤ *m* ∧ *y* ≤ *n* ∧ *x* > 0
λ (*x*, *y*). *if y* > 0 *then* (*x*, *y* − 1) *else* (*x* − 1, *n*)
λ (*x*, *y*). (*m* − *x*) ∗ (*n* + 1) + (*n* − *y*)
⟨*proof*⟩

lemma [*intro*]:
dp_consistency_heap_array_pair′ (*n* + 1) *fst snd id m* (*m* + 1) *mem*
⟨*proof*⟩

lemma [*intro*]:
dp_consistency_heap_array_pair_iterator (*n* + 1) *fst snd id m* (*m* + 1) *mem*
(λ (*x*, *y*). *if y* > 0 *then* (*x*, *y* − 1) *else* (*x* − 1, *n*))
(λ (*x*, *y*). (*m* − *x*) ∗ (*n* + 1) + (*n* − *y*))

$(\lambda\ (x,\ y).\ x \leq m \wedge y \leq n \wedge x > 0)$

⟨*proof*⟩

memoize_fun *min_ed_ix*$_h$: *min_ed_ix*
with_memory (**default_proof**) *dp_consistency_heap_array_pair_iterator*
where *size* = *n* + 1
and *key*1=*fst* :: *nat* × *nat* ⇒ *nat* **and** *key*2=*snd* :: *nat* × *nat* ⇒ *nat*
and *k*1=*m* :: *nat* **and** *k*2=*m* + 1 :: *nat*
and *to_index* = *id* :: *nat* ⇒ *nat*
and *mem* = *mem*
and *cnt* = λ (*x*, *y*). *x* ≤ *m* ∧ *y* ≤ *n* ∧ *x* > 0
and *nxt* = λ (*x*::*nat*, *y*). *if y* > 0 *then* (*x*, *y* − 1) *else* (*x* − 1, *n*)
and *sizef* = λ (*x*, *y*). (*m* − *x*) ∗ (*n* + 1) + (*n* − *y*)
monadifies (*heap*) *min_ed_ix.simps*

memoize_correct
⟨*proof*⟩

lemmas *memoized_empty* =
min_ed_ix$_h$.*memoized_empty*[*OF min_ed_ix*$_h$.*consistent_DP_iter_and_compute*[*OF min_ed_ix*$_h$.*crel*]]
lemmas *iter_heap_unfold* = *iter_heap_unfold*

end

end

3.6.8 Test Cases

abbreviation (*input*) *slice xs i j* ≡ *map xs* [*i*..<*j*]

lemma *min_ed_Nil*1: *min_ed* [] *ys* = *length ys*
⟨*proof*⟩

lemma *min_ed_Nil*2: *min_ed xs* [] = *length xs*
⟨*proof*⟩

lemma *min_ed_ix_min_ed*: *min_ed_ix xs ys m n* (*i*,*j*) = *min_ed* (*slice xs i m*) (*slice ys j n*)
⟨*proof*⟩

Functional Test Cases

definition *min_ed_list xs ys* = *min_ed_ix* (λ*i*. *xs*!*i*) (λ*i*. *ys*!*i*) (*length xs*) (*length ys*) (0,0)

lemma *min_ed_list ''madagascar'' ''bananas''* = 6
⟨*proof*⟩

definition *min_ed_ia xs ys* = (*let a* = *IArray xs*; *b* = *IArray ys*
in min_ed_ix (λ*i*. *a*!!*i*) (λ*i*. *b*!!*i*) (*length xs*) (*length ys*) (0,0))

lemma *min_ed_ia ''madagascar'' ''bananas''* = 6
⟨*proof*⟩

Extracting an Executable Constant for the Imperative Implementation

ground_function *min_ed_ix*$_h$'*_impl*: *min_ed_ix*$_h$'*.simps*
termination
⟨*proof*⟩

lemmas [*simp del*] = *min_ed_ix*$_h$'*_impl.simps min_ed_ix*$_h$'*.simps*

lemma *min_ed_ix*$_h$'*_impl_def*:
includes *heap_monad_syntax*
fixes *m n* :: *nat*
fixes *mem* :: *nat ref* × *nat ref* × *nat option array ref* × *nat option array ref*
assumes *mem_is_init*: *mem* = *result_of* (*init_state* (*n* + 1) *m* (*m* + 1)) *Heap.empty*
shows *min_ed_ix*$_h$'*_impl xs ys m n mem* = *min_ed_ix*$_h$' *xs ys m n mem*
⟨*proof*⟩

definition
iter_min_ed_ix xs ys m n mem = *iterator_defs.iter_heap*
(λ (*x*, *y*). *x* ≤ *m* ∧ *y* ≤ *n* ∧ *x* > 0)
(λ (*x*, *y*). *if y* > 0 *then* (*x*, *y* − 1) *else* (*x* − 1, *n*))
(*min_ed_ix*$_h$'*_impl xs ys m n mem*)

lemma *iter_min_ed_ix_unfold*[*code*]:
iter_min_ed_ix xs ys m n mem = (λ (*i*, *j*).
(*if i* > 0 ∧ *i* ≤ *m* ∧ *j* ≤ *n*
then do {
min_ed_ix$_h$'*_impl xs ys m n mem* (*i*, *j*);
iter_min_ed_ix xs ys m n mem (*if j* > 0 *then* (*i*, *j* − 1) *else* (*i* − 1, *n*))
}

else Heap_Monad.return ()))
⟨*proof*⟩

definition
min_ed_ix_impl xs ys m n i j = do {
mem ← (init_state (n + 1) (m::nat) (m + 1) ::
(nat ref × nat ref × nat option array ref × nat option array ref)
Heap);
iter_min_ed_ix xs ys m n mem (m, n);
min_ed_ix$_h$'_impl xs ys m n mem (i, j)
}

lemma *bf_impl_correct*:
min_ed_ix xs ys m n (i, j) = result_of (min_ed_ix_impl xs ys m n i j)
Heap.empty
⟨*proof*⟩

Imperative Test Case

definition
min_ed_ia$_h$ xs ys = (let a = IArray xs; b = IArray ys
in min_ed_ix_impl (λi. a!!i) (λi. b!!i) (length xs) (length ys) 0 0)

definition
test_case = min_ed_ia$_h$ ''madagascar'' ''bananas''

export_code *min_ed_ix* **in** *SML* **module_name** *Test*

code_reflect *Test* **functions** *test_case*

One can see a trace of the calls to the memory in the output

⟨*ML*⟩

end

3.7 Optimal Binary Search Trees

The material presented in this section just contains a simple and non-optimal version (cubic instead of quadratic in the number of keys). It can now be viewed to be superseded by the AFP entry *Optimal_BST*. It is kept here as a more easily understandable example and for archival purposes.

theory *OptBST*
imports
HOL−Library.Tree
HOL−Library.Code_Target_Numeral

../state_monad/State_Main
../heap_monad/Heap_Default
Example_Misc
HOL−Library.Product_Lexorder
HOL−Library.RBT_Mapping
begin

3.7.1 Function *argmin*

Function *argmin* iterates over a list and returns the rightmost element that minimizes a given function:

fun *argmin* :: (′*a* ⇒ (′*b*::*linorder*)) ⇒ ′*a list* ⇒ ′*a* **where**
argmin f (*x*#*xs*) =
(*if xs* = [] *then x else*
let m = *argmin f xs in if f x* < *f m then x else m*)

Note that *arg_min_list* is similar but returns the leftmost element.

lemma *argmin_forall*: *xs* ≠ [] ⟹ (⋀*x*. *x*∈*set xs* ⟹ *P x*) ⟹ *P* (*argmin f xs*)
⟨*proof*⟩

lemma *argmin_Min*: *xs* ≠ [] ⟹ *f* (*argmin f xs*) = *Min* (*f* ‘ *set xs*)
⟨*proof*⟩

3.7.2 Misc

lemma *upto_join*: ⟦ *i* ≤ *j*; *j* ≤ *k* ⟧ ⟹ [*i*..*j*−1] @ *j* # [*j*+1..*k*] = [*i*..*k*]
⟨*proof*⟩

lemma *atLeastAtMost_split*:
{*i*..*j*} = {*i*..*k*} ∪ {*k*+1..*j*} **if** *i* ≤ *k k* ≤ *j* **for** *i j k* :: *int*
⟨*proof*⟩

lemma *atLeastAtMost_split_insert*:
{*i*..*k*} = *insert k* {*i*..*k*−1} **if** *k* ≥ *i* **for** *i* :: *int*
⟨*proof*⟩

3.7.3 Definitions

context
fixes *W* :: *int* ⇒ *int* ⇒ *nat*
begin

fun *wpl* :: *int* ⇒ *int* ⇒ *int tree* ⇒ *nat* **where**

```
  wpl i j Leaf = 0
| wpl i j (Node l k r) = wpl i (k−1) l + wpl (k+1) j r + W i j

function min_wpl :: int ⇒ int ⇒ nat where
min_wpl i j =
  (if i > j then 0
   else min_list (map (λk. min_wpl i (k−1) + min_wpl (k+1) j + W i j)
[i..j]))
  ⟨proof⟩
termination ⟨proof⟩
declare min_wpl.simps[simp del]

function opt_bst :: int ⇒ int ⇒ int tree where
opt_bst i j =
  (if i > j then Leaf else argmin (wpl i j) [⟨opt_bst i (k−1), k, opt_bst (k+1)
j⟩. k ← [i..j]])
  ⟨proof⟩
termination ⟨proof⟩
declare opt_bst.simps[simp del]
```

3.7.4 Functional Memoization

```
context
  fixes n :: nat
begin

context fixes
  mem :: nat option array
begin

memoize_fun min_wpl_T: min_wpl
   with_memory dp_consistency_heap_default where bound = Bound
(0, 0) (int n, int n) and mem=mem
  monadifies (heap) min_wpl.simps

context includes heap_monad_syntax begin
thm min_wpl_T'.simps min_wpl_T_def
end

memoize_correct
  ⟨proof⟩

lemmas memoized_empty = min_wpl_T.memoized_empty
```

end

context
  includes *heap_monad_syntax*
  notes [*simp del*] = *min_wpl$_T$'.simps*
begin

definition *min_wpl$_h$* ≡ λ *i j*. *Heap_Monad.bind* (*mem_empty* (*n* ∗ *n*)) (λ *mem*. *min_wpl$_T$' mem i j*)

lemma *min_wpl_heap*:
  min_wpl i j = *result_of* (*min_wpl$_h$ i j*) *Heap.empty*
  ⟨*proof*⟩

end

end

context includes *state_monad_syntax* **begin**

memoize_fun *min_wpl$_m$*: *min_wpl* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *min_wpl.simps*
thm *min_wpl$_m$'.simps*

memoize_correct
  ⟨*proof*⟩
print_theorems
lemmas [*code*] = *min_wpl$_m$.memoized_correct*

memoize_fun *opt_bst$_m$*: *opt_bst* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *opt_bst.simps*
thm *opt_bst$_m$'.simps*

memoize_correct
  ⟨*proof*⟩
print_theorems
lemmas [*code*] = *opt_bst$_m$.memoized_correct*

end

3.7.5 Correctness Proof

lemma *min_wpl_minimal*:
  inorder t = [*i..j*] ⟹ *min_wpl i j* ≤ *wpl i j t*

⟨*proof*⟩

lemma *opt_bst_correct*: *inorder* (*opt_bst i j*) = [*i..j*]
⟨*proof*⟩

lemma *wpl_opt_bst*: *wpl i j* (*opt_bst i j*) = *min_wpl i j*
⟨*proof*⟩

lemma *opt_bst_is_optimal*:
inorder t = [*i..j*] ⟹ *wpl i j* (*opt_bst i j*) ≤ *wpl i j t*
⟨*proof*⟩

end

3.7.6 Access Frequencies

Usually, the problem is phrased in terms of access frequencies. We now give an interpretation of *wpl* in this view and show that we have actually computed the right thing.

context
— We are given a range [*i..j*] of integer keys with access frequencies *p*. These can be thought of as a probability distribution but are not required to represent one. This model assumes that the tree will contain all keys in the range [*i..j*]. See *Optimal_BST* for a model with missing keys.
fixes *p* :: *int* ⇒ *nat*
begin

— The *weighted path path length* (or *cost*) of a tree.
fun *cost* :: *int tree* ⇒ *nat* **where**
cost Leaf = 0
| *cost* (*Node l k r*) = *sum p* (*set_tree l*) + *cost l* + *p k* + *cost r* + *sum p* (*set_tree r*)

— Deriving a weight function from *p*.
qualified definition *W* **where**
W i j = *sum p* {*i..j*}

— We will use this later for computing *W* efficiently.
lemma *W_rec*:
W i j = (*if j* ≥ *i then W i* (*j* − 1) + *p j else* 0)
⟨*proof*⟩
lemma *inorder_wpl_correct*:
inorder t = [*i..j*] ⟹ *wpl W i j t* = *cost t*

⟨*proof*⟩

The optimal binary search tree has minimal cost among all binary search trees.

lemma *opt_bst_has_optimal_cost*:
inorder t = [*i*..*j*] ⟹ *cost* (*opt_bst W i j*) ≤ *cost t*
⟨*proof*⟩

The function *min_wpl* correctly computes the minimal cost among all binary search trees:

- Its cost is a lower bound for the cost of all binary search trees
- Its cost actually corresponds to an optimal binary search tree

lemma *min_wpl_minimal_cost*:
inorder t = [*i*..*j*] ⟹ *min_wpl W i j* ≤ *cost t*
⟨*proof*⟩

lemma *min_wpl_tree*:
cost (*opt_bst W i j*) = *min_wpl W i j*
⟨*proof*⟩

An alternative view of costs. **fun** *depth* :: ′*a* ⇒ ′*a tree* ⇒ *nat extended* **where**
depth x Leaf = ∞
| *depth x* (*Node l k r*) = (*if x* = *k then* 1 *else min* (*depth x l*) (*depth x r*) + 1)

fun *the_fin* **where**
the_fin (*Fin x*) = *x* | *the_fin* _ = *undefined*

definition *cost′* :: *int tree* ⇒ *nat* **where**
cost′ t = *sum* (λ*x*. *the_fin* (*depth x t*) ∗ *p x*) (*set_tree t*)

lemma [*simp*]:
the_fin 1 = 1
⟨*proof*⟩

lemma *set_tree_depth*:
assumes *x* ∉ *set_tree t*
shows *depth x t* = ∞
⟨*proof*⟩

lemma *depth_inf_iff*:
depth x t = ∞ ⟷ *x* ∉ *set_tree t*
⟨*proof*⟩

lemma *depth_not_neg_inf*[*simp*]:
depth x t = −∞ ⟷ *False*
⟨*proof*⟩

lemma *depth_FinD*:
assumes *x* ∈ *set_tree t*
obtains *d* **where** *depth x t* = *Fin d*
⟨*proof*⟩

lemma *cost′_Leaf*[*simp*]:
cost′ Leaf = 0
⟨*proof*⟩

lemma *cost′_Node*:
distinct (*inorder* ⟨*l*, *x*, *r*⟩) ⟹
cost′ ⟨*l*, *x*, *r*⟩ = *sum p* (*set_tree l*) + *cost′ l* + *p x* + *cost′ r* + *sum p* (*set_tree r*)
⟨*proof*⟩
lemma *weight_correct*:
distinct (*inorder t*) ⟹ *cost′ t* = *cost t*
⟨*proof*⟩

3.7.7 Memoizing Weights

function *W_fun* **where**
W_fun i j = (*if i* > *j then* 0 *else W_fun i* (*j* − 1) + *p j*)
⟨*proof*⟩

termination
⟨*proof*⟩

lemma *W_fun_correct*:
W_fun i j = *W i j*
⟨*proof*⟩

memoize_fun W_m: *W_fun*
with_memory *dp_consistency_mapping*
monadifies (*state*) *W_fun.simps*

memoize_correct

⟨*proof*⟩

definition
compute_W n = *snd* (*run_state* (*State_Main.map*$_T$′ (λ*i*. *W*$_m$′ *i n*) [0..*n*])
Mapping.empty)

notation *W*$_m$.*crel_vs* (*crel*)

lemmas *W*$_m$*_crel* = *W*$_m$.*crel*[*unfolded W*$_m$.*consistentDP_def*, *THEN rel_funD*,
of (*m*, *x*) (*m*, *y*) **for** *m x y*, *unfolded prod.case*]

lemma *compute_W_correct*:
assumes *Mapping.lookup* (*compute_W n*) (*i*, *j*) = *Some x*
shows *W i j* = *x*
⟨*proof*⟩
include *state_monad_syntax app_syntax lifting_syntax*
⟨*proof*⟩

definition
min_wpl′ *i j* ≡
let
M = *compute_W j*;
W = (λ*i j*. *case Mapping.lookup M* (*i*, *j*) *of None* ⇒ *W i j* | *Some x* ⇒
x)
in min_wpl W i j

lemma *W_compute*: *W i j* = (*case Mapping.lookup* (*compute_W n*) (*i*, *j*)
of None ⇒ *W i j* | *Some x* ⇒ *x*)
⟨*proof*⟩

lemma *min_wpl*′*_correct*:
min_wpl′ *i j* = *min_wpl W i j*
⟨*proof*⟩

definition
opt_bst′ *i j* ≡
let
M = *compute_W j*;
W = (λ*i j*. *case Mapping.lookup M* (*i*, *j*) *of None* ⇒ *W i j* | *Some x* ⇒
x)
in opt_bst W i j

lemma *opt_bst*′*_correct*:
opt_bst′ *i j* = *opt_bst W i j*

⟨*proof*⟩

end

3.7.8 Test Case

Functional Implementations

lemma *min_wpl* (λ*i j*. *nat*(*i*+*j*)) 0 4 = 10
⟨*proof*⟩

lemma *opt_bst* (λ*i j*. *nat*(*i*+*j*)) 0 4 = ⟨⟨⟨⟨⟨⟩, 0, ⟨⟩⟩, 1, ⟨⟩⟩, 2, ⟨⟩⟩, 3, ⟨⟩⟩, 4, ⟨⟩⟩
⟨*proof*⟩

Using Frequencies

definition
list_to_p xs (*i*::*int*) = (*if i* − 1 ≥ 0 ∧ *nat* (*i* − 1) < *length xs then xs* ! *nat* (*i* − 1) *else* 0)

definition
*ex_p_*1 = [10, 30, 15, 25, 20]

definition
*opt_tree_*1 =
⟨
⟨
⟨⟩, 1::*int*, ⟨⟩⟩,
2,
⟨⟨⟩, 3, ⟨⟩⟩
⟩,
4,
⟨⟨⟩, 5, ⟨⟩⟩
⟩

lemma *opt_bst′* (*list_to_p ex_p_*1) 1 5 = *opt_tree_*1
⟨*proof*⟩

Imperative Implementation

code_thms *min_wpl*

definition *min_wpl_test* = *min_wpl*$_h$ (λ*i j*. *nat*(*i*+*j*)) 4 0 4

code_reflect *Test* **functions** *min_wpl_test*

⟨*ML*⟩

end

3.8 Longest Common Subsequence

theory *Longest_Common_Subsequence*
imports
HOL−Library.Sublist
HOL−Library.IArray
HOL−Library.Code_Target_Numeral
HOL−Library.Product_Lexorder
HOL−Library.RBT_Mapping
../state_monad/State_Main
begin

3.8.1 Misc

lemma *finite_subseq*:
finite {*xs*. *subseq xs ys*} (**is** *finite ?S*)
⟨*proof*⟩

lemma *subseq_singleton_right*:
subseq xs [*x*] = (*xs* = [*x*] ∨ *xs* = [])
⟨*proof*⟩

lemma *subseq_append_single_right*:
subseq xs (*ys* @ [*x*]) = ((∃ *xs′*. *subseq xs′ ys* ∧ *xs* = *xs′* @ [*x*]) ∨ *subseq xs ys*)
⟨*proof*⟩

lemma *Max_nat_plus*:
Max (((+) *n*) ‘ *S*) = (*n* :: *nat*) + *Max S* **if** *finite S S* ≠ {}
⟨*proof*⟩

3.8.2 Definitions

context
fixes *A B* :: *′a list*
begin

fun *lcs* :: *nat* ⇒ *nat* ⇒ *nat* **where**
lcs 0 _ = 0 |
lcs _ 0 = 0 |

lcs (Suc i) (Suc j) = (if A!i = B!j then 1 + lcs i j else max (lcs i (j + 1)) (lcs (i + 1) j))

definition *OPT i j = Max {length xs | xs. subseq xs (take i A) ∧ subseq xs (take j B)}*

lemma *finite_OPT*:
finite {xs. subseq xs (take i A) ∧ subseq xs (take j B)} (**is** *finite ?S*)
⟨*proof*⟩

3.8.3 Correctness Proof

lemma *non_empty_OPT*:
{xs. subseq xs (take i A) ∧ subseq xs (take j B)} ≠ {}
⟨*proof*⟩

lemma *OPT_0_left*:
OPT 0 j = 0
⟨*proof*⟩

lemma *OPT_0_right*:
OPT i 0 = 0
⟨*proof*⟩

lemma *OPT_rec1*:
OPT (i + 1) (j + 1) = 1 + OPT i j (**is** *?l = ?r*)
if *A!i = B!j i < length A j < length B*
⟨*proof*⟩

lemma *OPT_rec2*:
OPT (i + 1) (j + 1) = max (OPT i (j + 1)) (OPT (i + 1) j) (**is** *?l = ?r*)
if *A!i ≠ B!j i < length A j < length B*
⟨*proof*⟩

lemma *lcs_correct′*:
OPT i j = lcs i j **if** *i ≤ length A j ≤ length B*
⟨*proof*⟩

theorem *lcs_correct*:
Max {length xs | xs. subseq xs A ∧ subseq xs B} = lcs (length A) (length B)
⟨*proof*⟩

end

3.8.4 Functional Memoization

context
  fixes *A B* :: *'a iarray*
begin

fun *lcs_ia* :: *nat* ⇒ *nat* ⇒ *nat* **where**
  lcs_ia 0 _ = 0 |
  lcs_ia _ 0 = 0 |
  lcs_ia (*Suc i*) (*Suc j*) =
    (*if A*!!*i* = *B*!!*j then* 1 + *lcs_ia i j else max* (*lcs_ia i* (*j* + 1)) (*lcs_ia* (*i* + 1) *j*))

lemma *lcs_lcs_ia*:
  lcs xs ys i j = *lcs_ia i j* **if** *A* = *IArray xs B* = *IArray ys*
  ⟨*proof*⟩

memoize_fun *lcs*$_m$: *lcs_ia* **with_memory** *dp_consistency_mapping* **monadifies** (*state*) *lcs_ia.simps*

memoize_correct
  ⟨*proof*⟩

lemmas [*code*] = *lcs*$_m$.*memoized_correct*

end

3.8.5 Test Case

definition *lcs*$_a$ **where**
  lcs$_a$ *xs ys* = (*let A* = *IArray xs*; *B* = *IArray ys in lcs_ia A B* (*length xs*) (*length ys*))

lemma *lcs*$_a$*_correct*:
  lcs xs ys (*length xs*) (*length ys*) = *lcs*$_a$ *xs ys*
  ⟨*proof*⟩

value *lcs*$_a$ *''ABCDGH'' ''AEDFHR''*

value *lcs*$_a$ *''AGGTAB'' ''GXTXAYB''*

end

theory *All_Examples*
imports
Bellman_Ford
Knapsack
Counting_Tiles
CYK
*Min_Ed_Dist*0
OptBST
Longest_Common_Subsequence
begin

end

References

[1] J. M. Kleinberg and É. Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[2] S. Wimmer, S. Hu, and T. Nipkow. Verified memoization and dynamic programming. In J. Avigad and A. Mahboubi, editors, *ITP 2018, Proceedings*, Lecture Notes in Computer Science. Springer, 2018.